Programmēšanas pamati ar valodu *Python*

Saturs īsi

Saturs detalizēti

1. Pirms sākt

1.1. Par šo mācību materiālu

Python programmēšanas pamatu mācību materiāls latviešu valodā izstrādāts valodas *Python* versijai 3. Pievienotie piemēri darbināti uz Windows 10 ar standarta *Python* implementāciju v.3.8 (https://www.python.org/). Labākai uzskatāmībai piemēru demonstrēšanā izmantota arī izstrādes vide *Wing 101* (http://wingware.com/).

1.2. Kāpēc *Python?*

Python ir salīdzinoši **ļoti** augsta līmeņa programmēšanas valoda, kas (ne jau gluži tieši, bet) nozīmē, ka tās pašas funkcionalitātes uzrakstīšanai nepieciešams mazāks rindiņu skaits nekā zemāka līmeņa programmēšanas valodā (piemēram, C++) un pirmkods (kaut kādā nozīmē) ir vieglāk saprotams. Valodu *Python* mēdz saukt par prototipēšanas valodu – jo tajā var viegli un ātri pārbaudīt „lietas", tāpēc tā ir populāra pētnieku vidū, bet tas nenozīmē, ka *Python* nav noderīgs „normālu" programmu rakstīšanai, turklāt tieši pēdējos gados *Python* popularitāte ir strauji augusi.

Python pozitīvās īpašības:

- Elegants un kompakts pieraksts;
- Produktīvs programmēšanas darbs;
- Viegli apgūt valodas pamatus.

Python negatīvās īpašības:

- Apjomīgas programmas vispārīgā gadījumā strādā salīdzinoši lēni;
- Nopietnu programmu (grafiskā) saskarne var nebūt pietiekoši elastīga/eleganta.

Python kā pirmā programmēšanas valoda:

- **Pozitīvā** puse ir tā, ka, pateicoties augtajam līmenim, nav jādomā par fiziskām un tehniskām lietām un var veltīt visu potenciālu idejas/algoritma veidošanai, turklāt „to pašu" var noprogrammēt ātrāk un īsāk.
- **Negatīvā** puse ir, ka valodas „augstais līmenis" daudzos gadījumos prasa diezgan daudz abstrakcijas un papildus jēdzienu, kas var izrādīties nopietns izaicinājums.

1.3. Programmēšanas valodas *Python izcelšanās*

Valodu *Python* 1991. gadā izstrādājis nīderlandiešu programmētājs **Guido van Rosums** (*Guido van Rossum*). Python ir augsta līmeņa universāla pielietojuma valoda ar dinamisku tipu sistēmu un automātisku atmiņas pārvaldību. Valodas nosaukums cēlies no televīzijas seriāla *„Monty Python's Flying Circus". Python* ir interpretējama programmēšanas valoda (tātad, netiek veidots speciāls izpildāmais kods, bet programma tiek darbināta „pa taisno"). Ļoti augstā līmeņa un interpretējamības dēļ *Python* sauc arī par „skriptu" valodu. Divas „lielās" *Python* versijas (atzari) ir 2.0 (sākotnēji izlaista 2000. gadā, bet joprojām vairākās sistēmās (piemēram, dažādos *Linux*) kā noklusētā versija, kā arī joprojām populāra pētnieku vidū) un 3.0 (sākotnēji izlaista 2008. gadā), uz kuras bāzes izstrādāts šis mācību materiāls (konkrētāk, *Python* 3.8).

1.4. Python uzstādīšana

Lai strādātu (programmētu) valodā *Python*, nepieciešams uzstādīt

- pašu *Python* interpretatoru (kompilatoru), kas darbina *Python* programmas,
- izstrādes vidi (*IDE*) ērtākai programmas koda veidošanai.

Tipiska interpretatora izvēle ir https://www.python.org/, kur ir pieejamas gan *Python2*, gan *Python3* jaunākās versijas. (Šajā mācību materiālā izmantots *Python3*.)

Izstrādes vides pieejamas daudz lielākā klāstā – ir gan komerciālās, gan brīvās alternatīvas. Šajā mācību materiālā tiks izmantots *Wing*.

Parasti vispirms tiek uzstādīts *Python* kā programmēšanas valoda un tikai tad, pieņemot, ka *Python* jau ir uzstādīts, tiek uzstādīta arī izvēlētā izstrādes vide (*IDE*). Veiksmīgas *IDE* uzstādīšanas gadījumā, uzstādīšanas beigās parasti ir dialoglogs, kas apliecina, ka izstrādes vide ir atpazinusi, ka uz datora ir uzstādīts *Python* un paņēmusi to izmantošanā.

Šī mācību materiāla apgūšanai iesaku divus iespējamos alternatīvos *Python* (kompilatora un izstrādes vides) uzstādīšanas variantus (bet tas nenozīmē, ka nevar lietot citus):

- Ja dators ir pietiekoši jaudīgs, tad ērta būs *Anaconda* platformas uzstādīšana, kas ietver sevī gan kompilatoru, gan izstrādes vidi *Spyder*; https://www.anaconda.com/
- Ja dators nav tik spēcīgs, tad
    - vispirms uzstādīt *Python* kompilatoru: https://www.python.org/
    - pēc tam izstrādes vidi *Wing 101*: http://wingware.com/

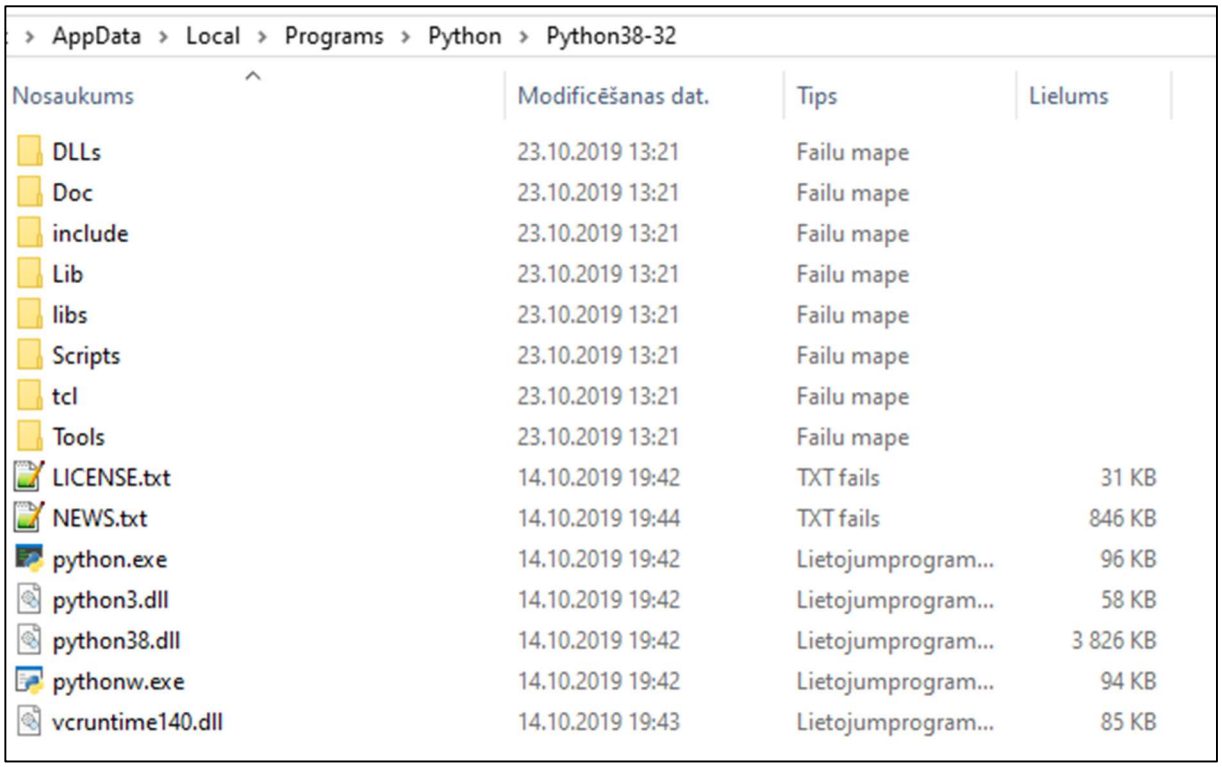
Tālāk īsi aprakstīta otrās alternatīvas komponentu uzstādīšana, jo kursa materiāli tiks ilustrēti, izmantojot *Wing 101* izstrādes vidi.

1.4.1. *Python* interpretatora uzstādīšana

Python ielādēšanas lapa https://www.python.org/ izskatās šādi:

Uzstādīšanas, kas ir pilnīgi „parasta", rezultātā parādās *Python* programmu grupa:

Visa ar *Python* saistītā „saimniecība" ievietojas kādā direktorijā, kuras nosaukums ir saistīts ar *Python* versiju:



Šī piemēra kontekstā pats *Python* kompilatora fails ir *python.exe*.

1.4.2. *Python* integrētās izstrādes vides *Wing 101* uzstādīšana

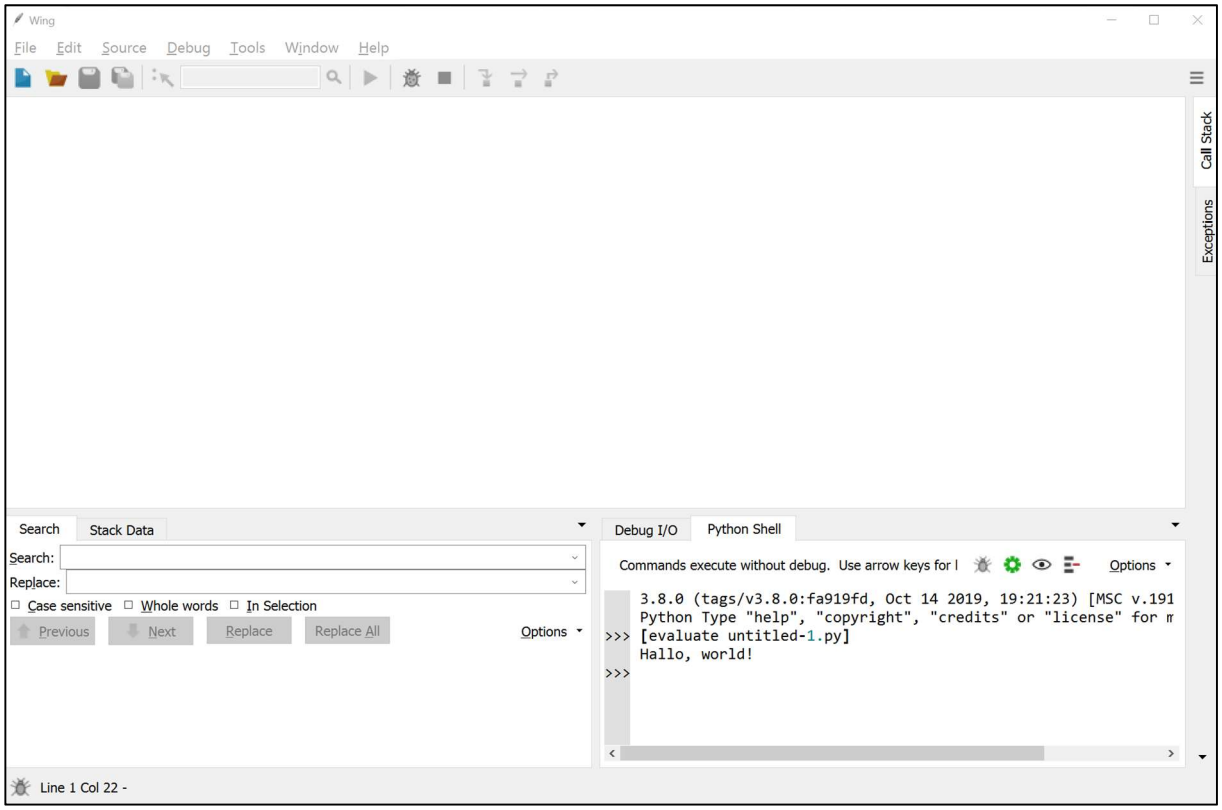
Python programmas var rakstīt arī, neizmantojot nekādus papildus līdzekļus, tomēr ērtumam un produktivitātei parasti kāda integrētās izstrādes vide (*IDE*) ir vēlama. Šajā mācību materiālā vide *Wing 101* izvēlēta tās vienkāršības un vairākplatformu atbalsta dēļ un tāpēc, ka tā ir par brīvu.

Instalēšanas failu var ielādēt no vietnes:

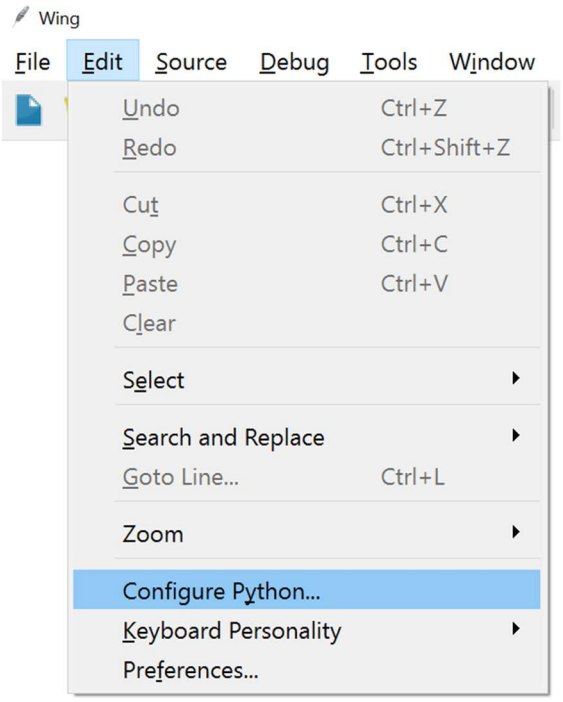
http://wingware.com/downloads/wing-101

izvēloties savai operētājsistēmai atbilstošo variantu:

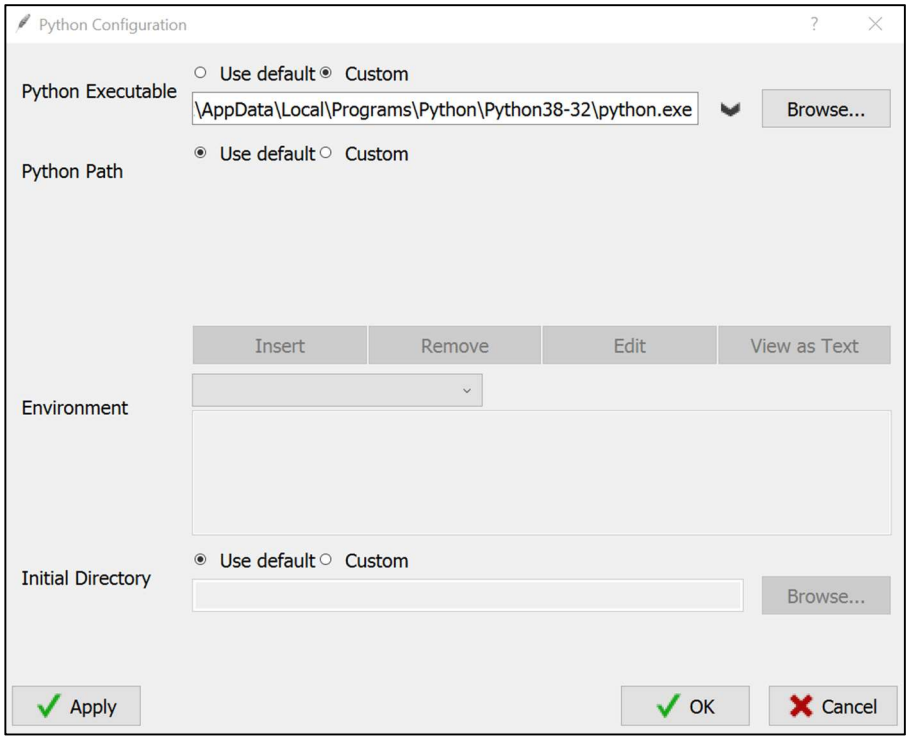
Pēc instalēšanas pirmo reizi atverot *Wing*, var redzēt, ka tas ir atpazinis, ka uz mana datora tiek lietots *Python* 3.8.0:



Vairāku *Python* kompilatoru gadījumā, iespējams izvēlēties citu kompilatoru, ar kuru darbosies *Wing (Edit--Configure Python)*:



Tad var izvēlēties citu kompilatoru, norādot ceļu uz to (*Python Executable*):



2. Pirmā programma *Python*

2.1. Kas ir *Python* programma, un kā tā strādā

Python programma vienkāršākajā gadījumā ir teksta fails (vai teksta fragments), bet vispārīgā gadījumā – viena vai vairāku teksta failu kopums.

Python programmu darbina *Python* interpretators (sk. 1.4.1) vai nu tiešā veidā vai arī pastarpināti caur integrēto izstrādes vidi, piemēram, nospiežot pogu „Run" utml.

Darbinātā programma pēc būtības var izskatīties kā jebkura cita lietojumprogramma, tomēr šī mācību materiāla ietvaros neizmantosim nekādu grafisku lietotāja saskarni, bet gan iztiksim ar **konsoles** tipa saskarni – t.i. teksta lodziņu interaktīvā režīmā, koncentrējoties uz dažādu vispārīgas programmēšanas jēdzienu un tehniku apgūšanu.

Tātad, rakstīsim programmas, kas no lietotāja viedokļa ļauj ievadīt tekstu un izdrukā tekstu konsoles lodziņā lineārā veidā (rindu pēc rindas).

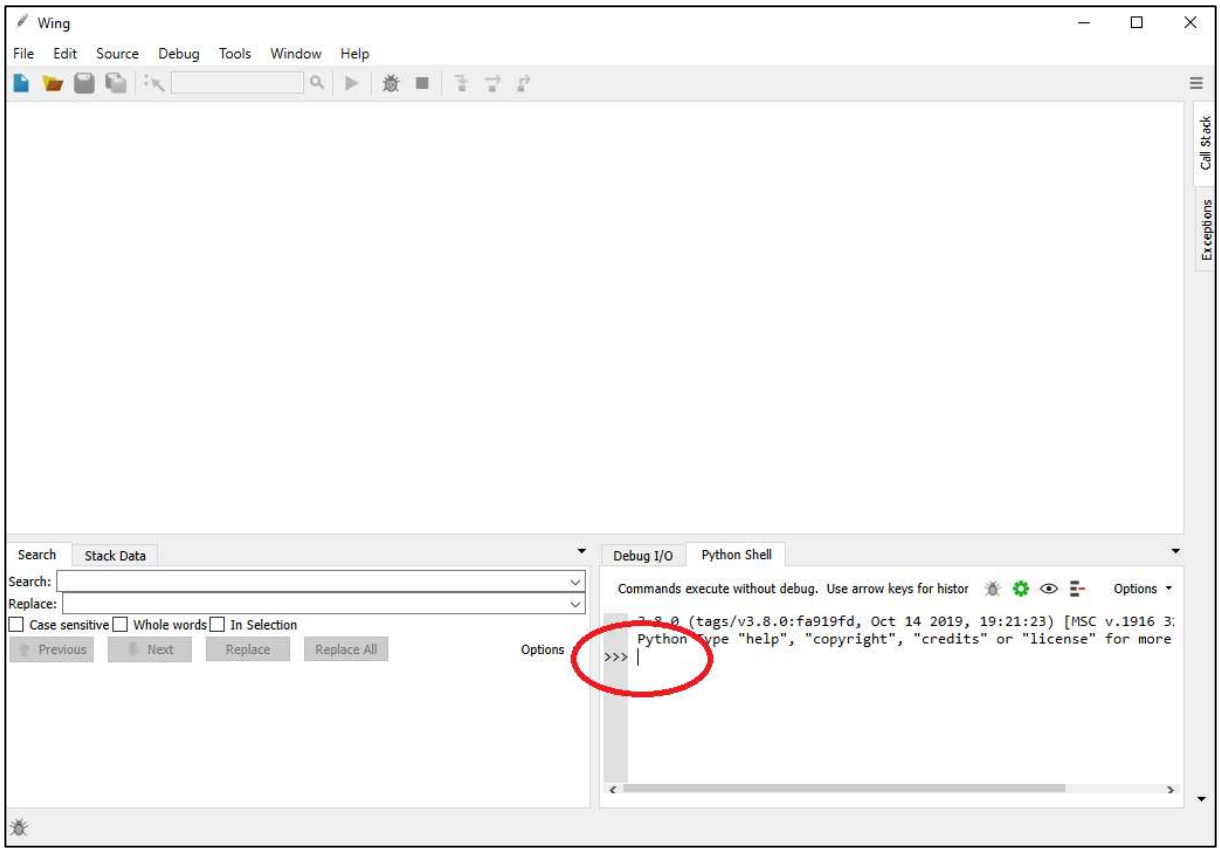
2.2. Python komandrindu logs – esošu programmu darbināšanai un *Python* izmantošanai pat bez programmas rakstīšanas

Python interpretatoram dažādos veidos līdzi nāk t.s. interaktīvā vide – speciāls „pitona" (komandrindu) lodziņš, kurā ievadīt komandas un nolasīt rezultātus teksta formā. Turklāt komandu ievadīšana var izpausties gan kā esošas programmas vai fragmenta izsaukšana, gan programmas fragmenta uzrakstīšana „pa tiešo".

Komandas rakstīšana komandrindu logā visur ir līdzīga un izpaužas kā noteiktas komandas uzrakstīšana, kuras beigās seko ENTER, kas nosūta komandu izpildīšanai.

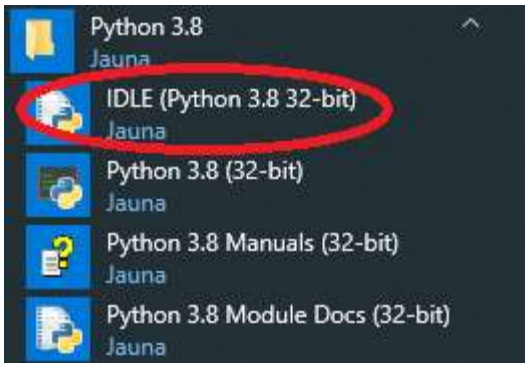
2.2.1. Komandrindu logs *Wing 101* vidē

Atverot izstrādes vidi *Wing 101*, komandrindu lodziņš parādās augšējā daļā, bet komandrinda seko aiz simboliem ">>>". Komandrindas aktivitāti nosaka mirgojošs kursors:

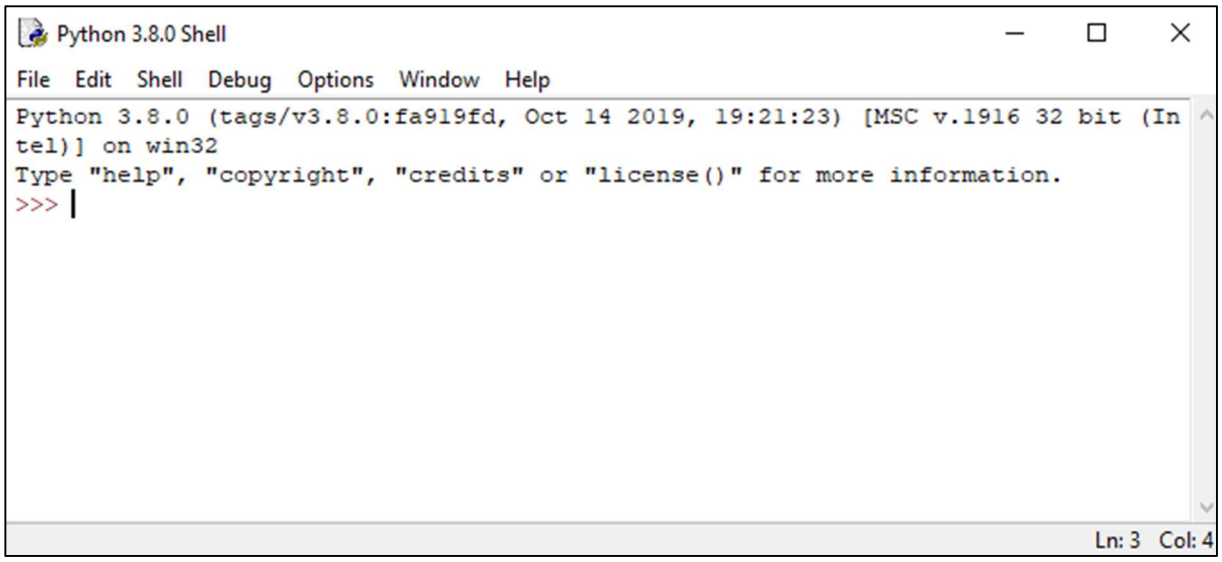


2.2.2. Komandrindu logs *IDLE* vidē

Arī standarta *Python* interpretatoram nāk līdzi iebūvēta vienkārša izstrādes vide ar nosaukumu *IDLE*:



kurai arī ir šāds interaktīvs lodziņš:



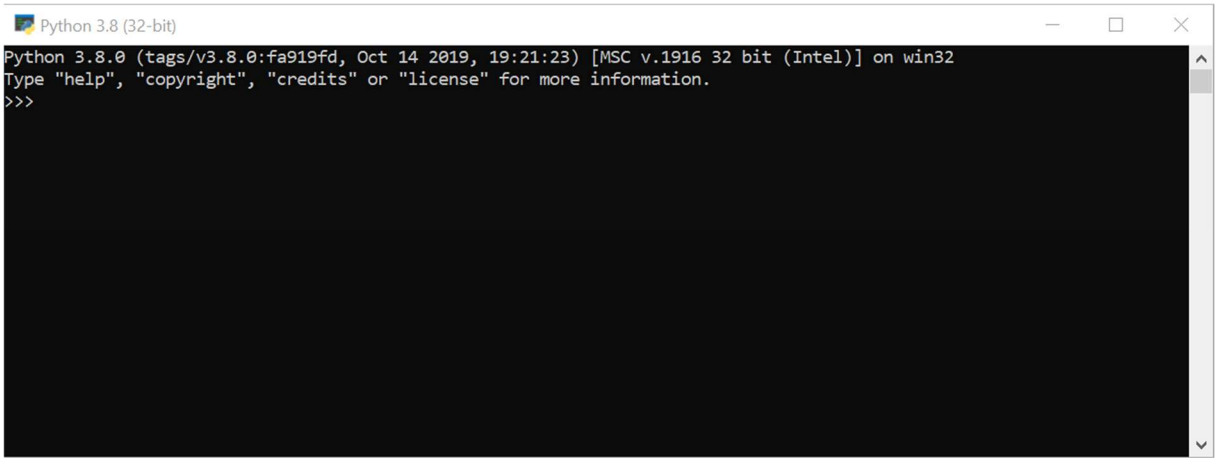
2.2.3. *Python* „tiešais" komandrindu logs

Izsaucot *Python* „tiešā" veidā kādā no veidiem:

- Windows sistēmā no programmu izvēlnes:

- Komandrindā „pa taisno" izsaucot *Python* interpretatoru, kas šī mācību materiāla kontekstā (uz Windows) būtu, izsaucot "../python.exe", bet *Linux* sistēmās tipiski vienkārši "python3" (jo "python" pēc noklusējuma parasti nozīmē Python 2. versijas izsaukumu – mēs izmantojam 3. versiju!)

tiek iegūts apmēram šāds logs:

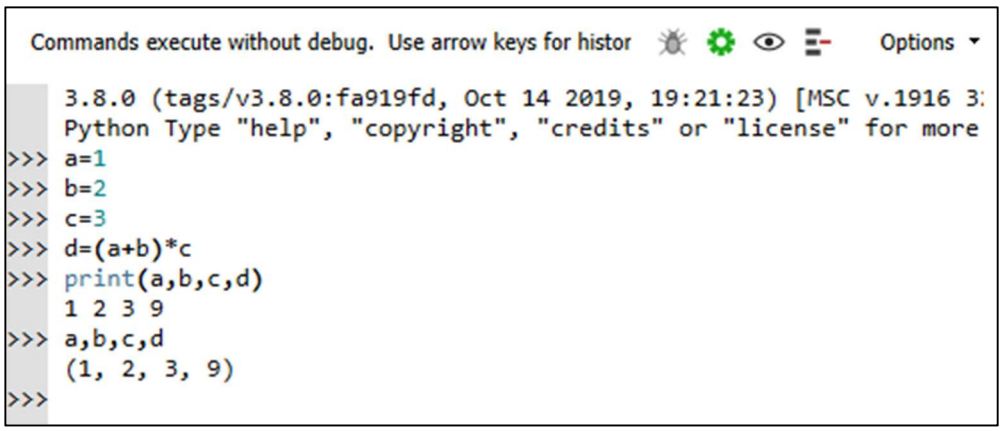


2.2.4. Pirmā *Python* darbināšana – komandas ievade komandrindu logā

Turpmāk piemēros tiks izmantots *Wing 101* komandrindu logs, bet tas līdzīgi darbojas arī citos.

Tas, ko tiešā veidā var ievadīt komandrindu logā bez programmas rakstīšanas failā, ir dažādas izteiksmes (arī ne tikai skaitliskas), piemēram, 3+5, lai noskaidrotu, ka tas ir 8:

Var veikt arī sarežģītāku izteiksmju aprēķinu, izmantojot vairāku soļu pierakstu ar starprezultātu noglabāšanu mainīgajos:



Bet vispār tur var rakstīt arī pilnu programmas tekstu, bet tas komandrindā pa rindiņai būtu neērti ievadāms.

2.3. Beidzot arī pirmā programma *Python* – Hello, World!

Daudzu programmēšanas valodu apmācīšanu parasti sāk ar „Hello, World!" programmu, kas tiek izmantota arī šeit un parādīta Att. 2-1. Šī programma, kā jau redzams tās darbības piemērā, uz ekrāna izdrukā tekstu "Hello, World!". Valodā *Python* šī programma sastāv no 1 rindiņas.

Sākot ar šo programmu un turpmāk materiālā, programmas vai to fragmenti tiks marķēti ar biezu svītru programmas koda kreisajā pusē, pēc tam var sekot **programmas darbības piemērs**, kurš marķēts ar trīskāršo svītru kreisajā pusē. Ja programmas darbības piemērā parādās lietotāja ievads no klaviatūras, tas tiek atzīmēts treknināti (*bold*).

```
print('Hello, World!')
```

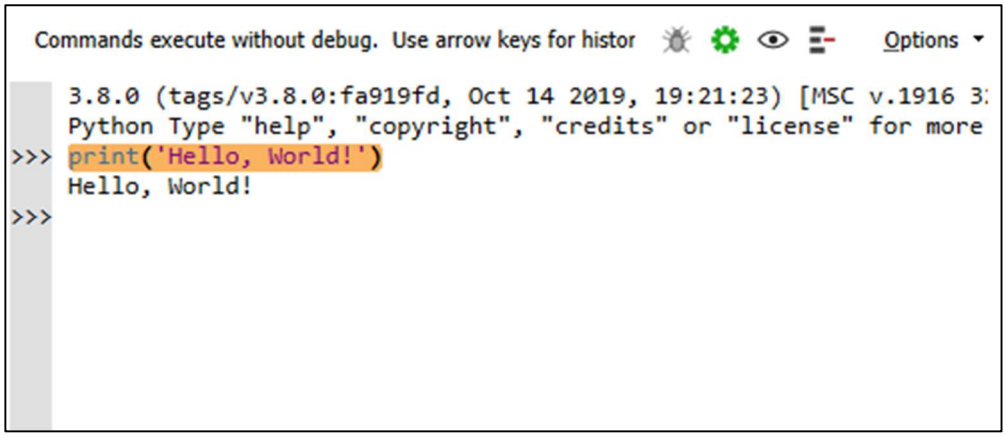
```
Hallo, World!
```

Att. 2-1. Programma „Hello, World!" valodā *Python* un tās darbības rezultāts

Turpmāk vairāki varianti, kā šo vienkāršo programmu pierakstīt un kā palaist.

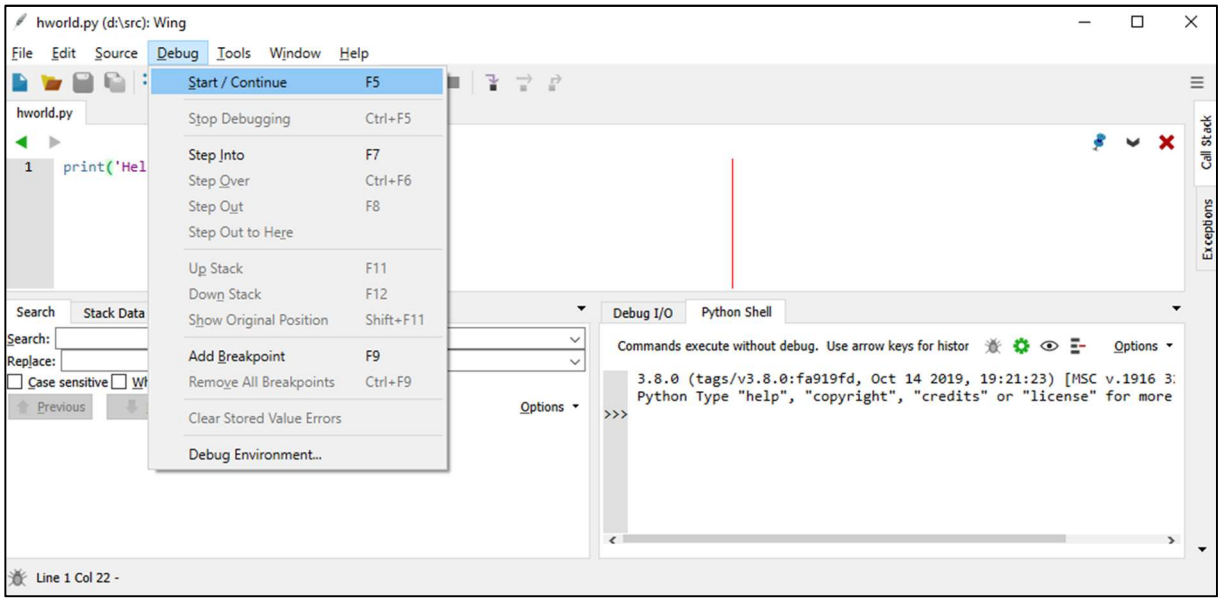
2.3.1. Programmas teksta ievietošana tieši *Python* komandrindā

Programmas tekstu var nerakstīt failā, bet pa taisno ievietot *Python* komandrindā. Ja programmā ir vairākas rindas, tad tas jādara pa vienai rindai.

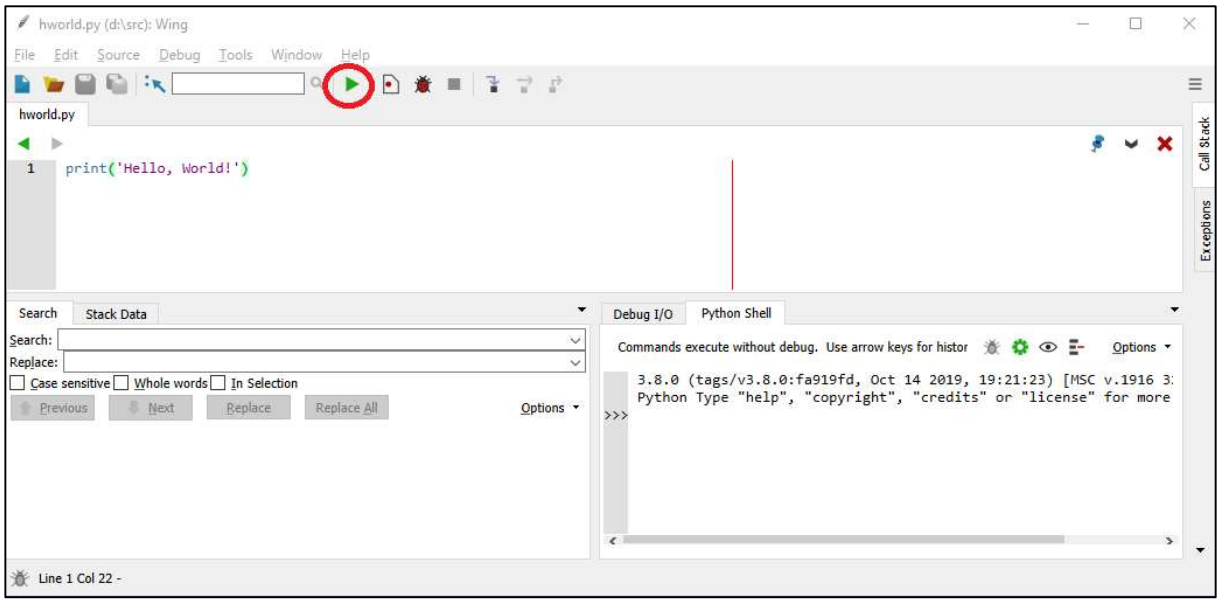


2.3.2. Programmas faila izpildīšana *Wing 101*

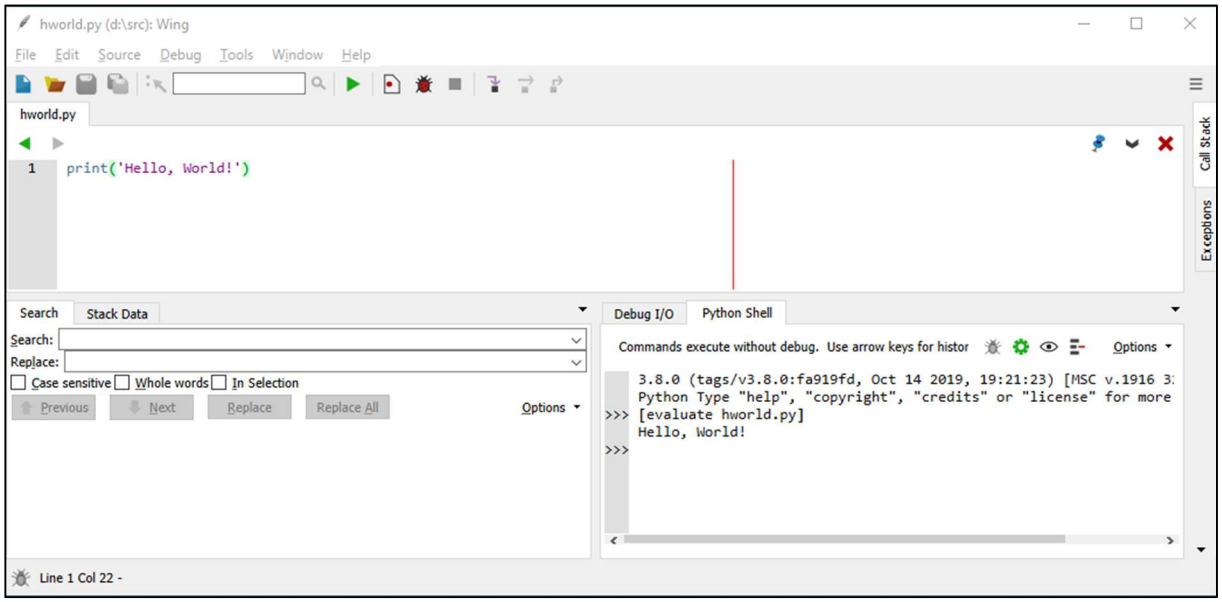
Programmu izpilda, izsaucot attiecīgu komandu. Atkarībā no izmantotās izstrādes vides, tā var būt izvēlnes komanda (šeit: kopā ar atkļūdošanu (*debug*)):



un/vai poga:

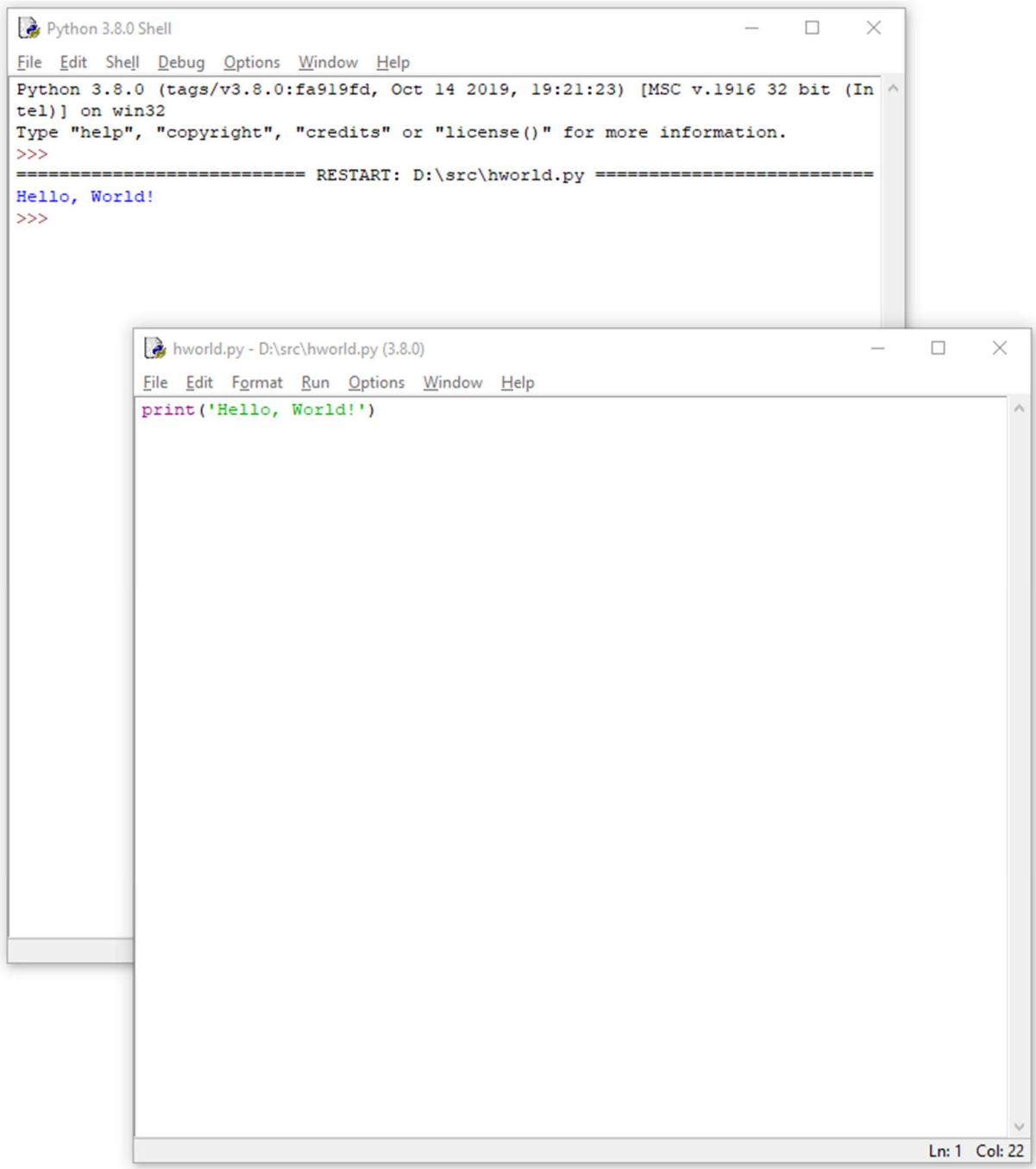


Programma nostrādā, izvadot rezultātu interaktīvajā lodziņā:



2.3.3. Programmas faila izpildīšana vidē *IDLE*

IDLE vidē, lai programmu izpildītu, jāatver fails (*File--Open*), kurš atveras jaunā logā, kurā tad jāizsauc komanda *Run--Run module*:



2.3.4. Programmas faila izpildīšana ārpus izstrādes vides

Python programmas failu var palaist pa taisno, (operētājsistēmas) komandrindā izsaucot interpretatoru (*../python.exe*) un tam kā papildus informāciju (parametru) padodot izpildāmo *Python* failu (mūsu gadījumā: *D:/src/hworld.py*):

Turpmāk mācību materiālā ērtības un uzskatāmības dēļ programmu darbināšanai tiks izmantota **tikai Wing 101 vide**.

Programmu var izsaukt vai nu kā failu, vai kā moduli (papildus parametrs *–m*, un failam nav *.py* paplašinājuma):

```
D:\src>python hworld.py
Hello, World!

D:\src>python -m hworld
Hello, World!
```

2.4. Otrā programma – ar lietotāja ievadu

2.4.1. Programma un tās darbināšana

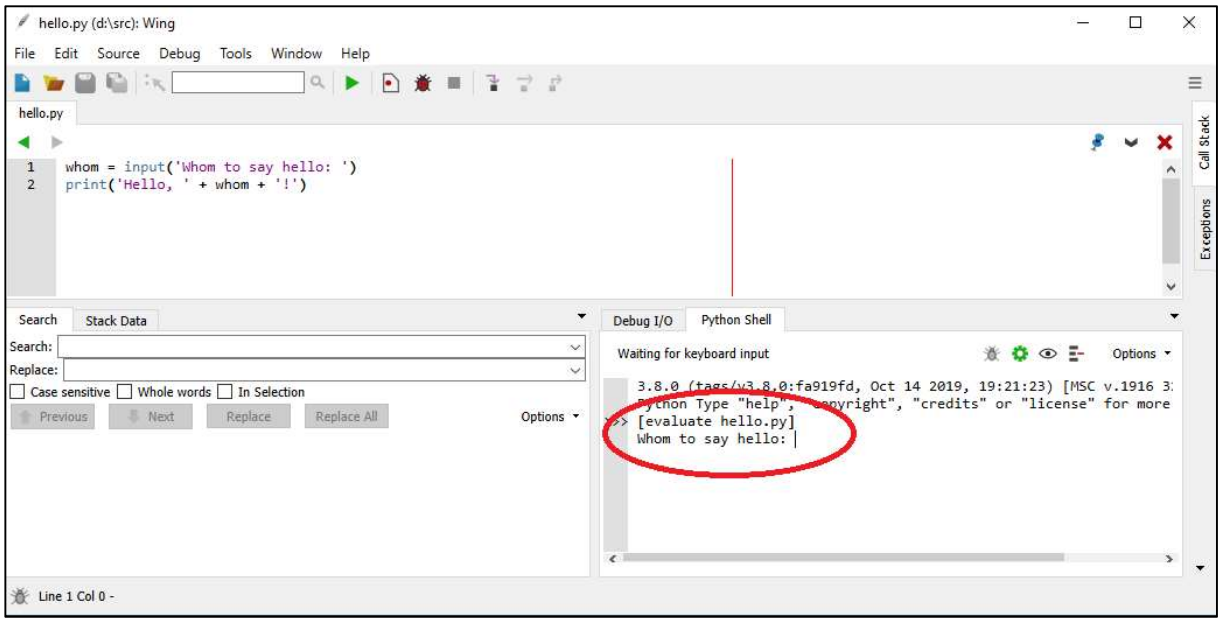
Otrajā programmā, tas, ko pasveicināt, ir ne obligāti pasaule, bet tas, ko nosaka lietotājs, ievadot attiecīgu tekstu:

```
whom = input('Whom to say hello: ')
print('Hallo, ' + whom + '!')
```

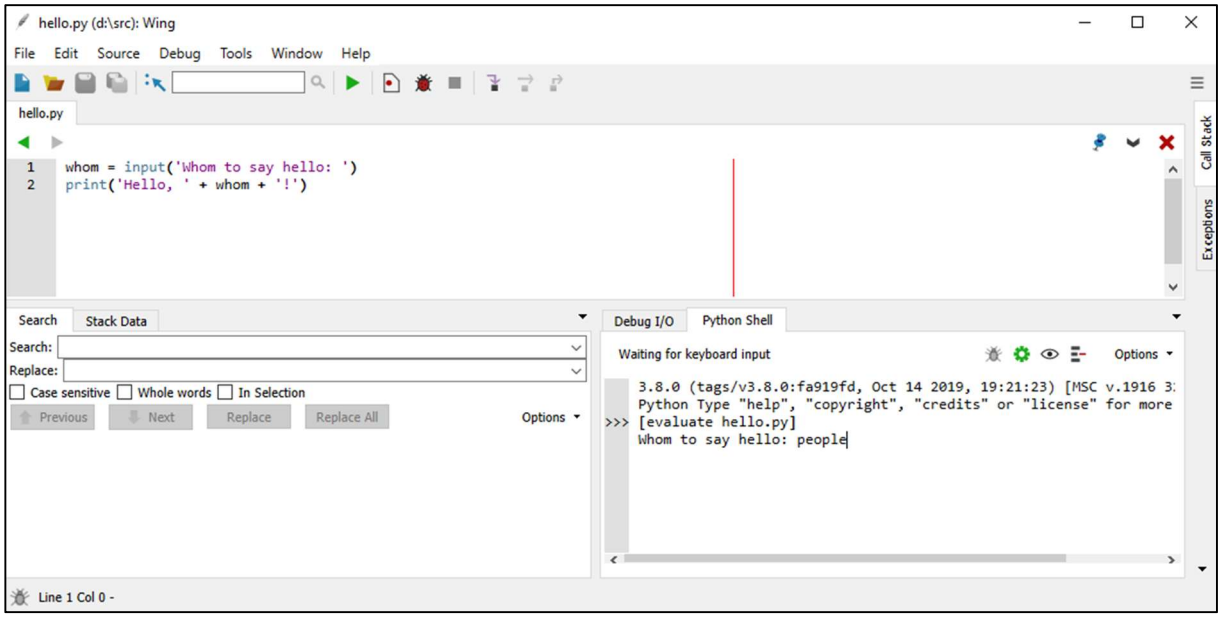
```
Whom to say hello: people
Hello, people!
```

Att. 2-2. Programma „Hello" un tās darbināšanas piemērs (trekināti – lietotāja ievads)

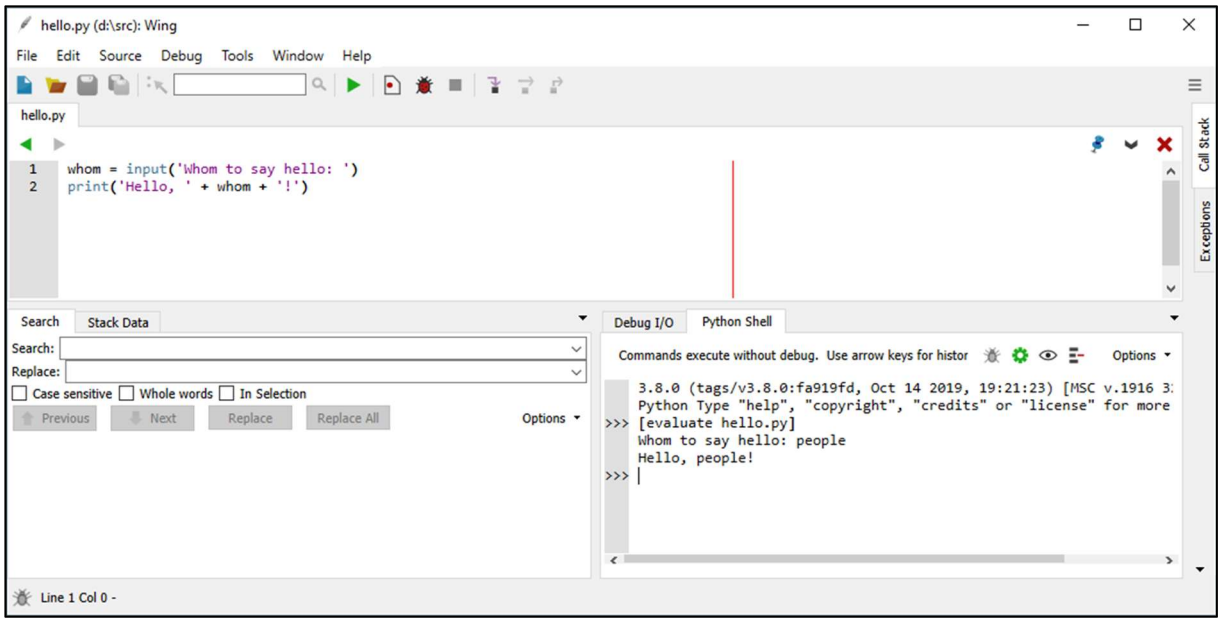
Sākumā darbinot programmu, tā apstājas pusceļā, gaidot lietotāja ievadu vietā, kur interaktīvajā lodziņā mirgo kursors (citas izstrādes vides piedāvā speciālu ievades lodziņu):



Tikai pēc prasītā vārda ievades (ko nodrošina funkcija (komanda) *input*, programmas darbība atbilstoši turpinās:



Tiek iegūts rezultāts:



2.4.2. Par programmas uzrakstīšanu

Kā jau tika minēts, programmu var rakstīt, neizmantojot speciālu izstrādes vidi, piemēram ar *Notepad* vai citu teksta redaktoru. Vairāki universālie teksta redaktori pat piedāvā t.s. programmas sintaktisko iekrāsošanu, kas palīdz uztvert programmas būtību.

Tomēr specializētas izstrādes vides (piemēram *Wing*) izmantošana ir vēlama vismaz šādu papildus priekšrocību dēļ:

- ērtāka vairāku failu programmu apkalpošana,
- automātiska daļēju vārdu pabeigšana un priekšāteikšana.

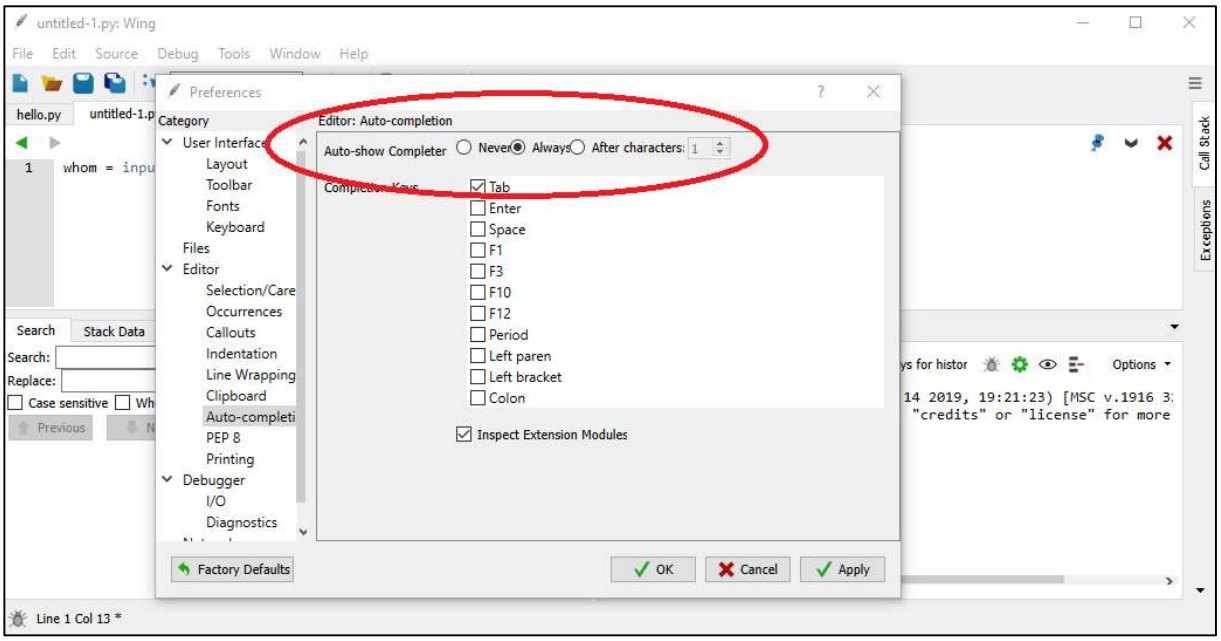
Īsi aplūkosim, kā otro lietu, rakstot programmu *hello.py*, dara *Wing*.

Pirms sākt rakstīt programmu, atcerēsimies, ka *Python* ir reģistrjūtīga (*case sensitive*) valoda, kas nozīmē, ka, veidojot nosaukumus (piemēram, mainīgo vārdus) tiek atšķirti lielie un mazie burti.

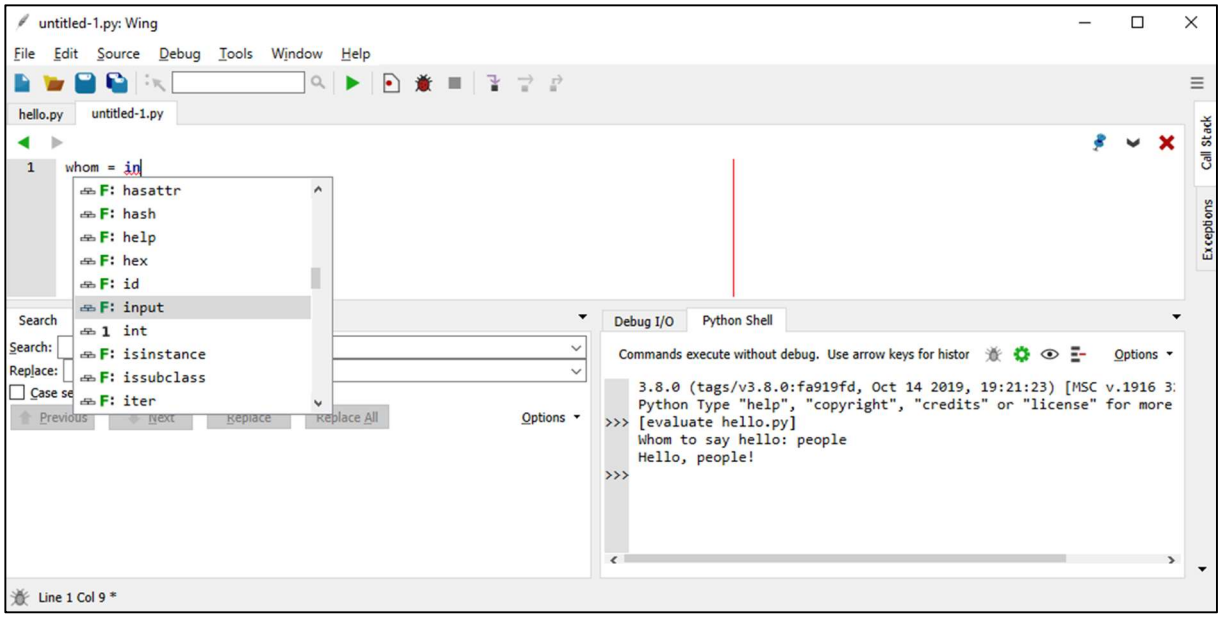
Programmā „Hello" tiek lietots mainīgais *whom*, kurā tiek glabāts starprezultāts – teksts, kuru lietotājs ievada no klaviatūras (mēs *whom* vietā varētu lietot arī citu vārdu). Bez tam tiek lietotas iebūvētās funkcijas *input* un *print*, kas katra ir izmantojama savā noteiktā veidā.

Sākam rakstīt jau iepriekš zināmo programmas *hello.py* tekstu, skatoties, kā *Wing* mums palīdz to darīt, atgādinot un sakot priekšā.

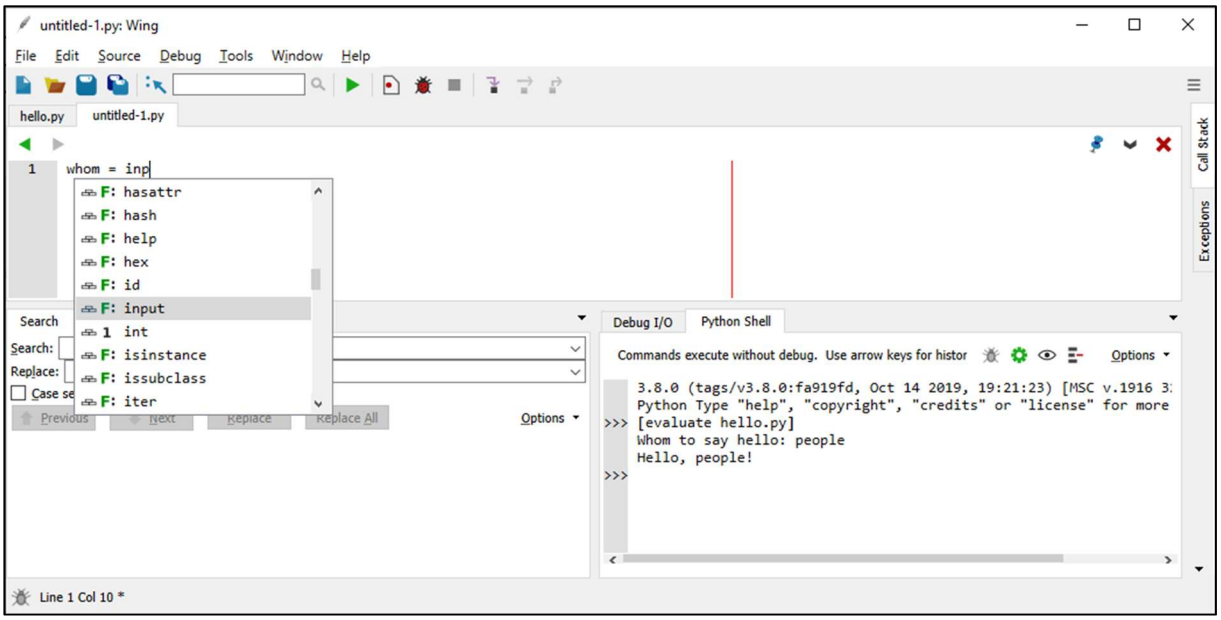
Lai nodrošinātu, ka tiek "teikts priekšā", *Wing 101* vispirms jāuzstāda, ka priekšā teikšana tiek aktivizēta (*Edit--Preferences*):



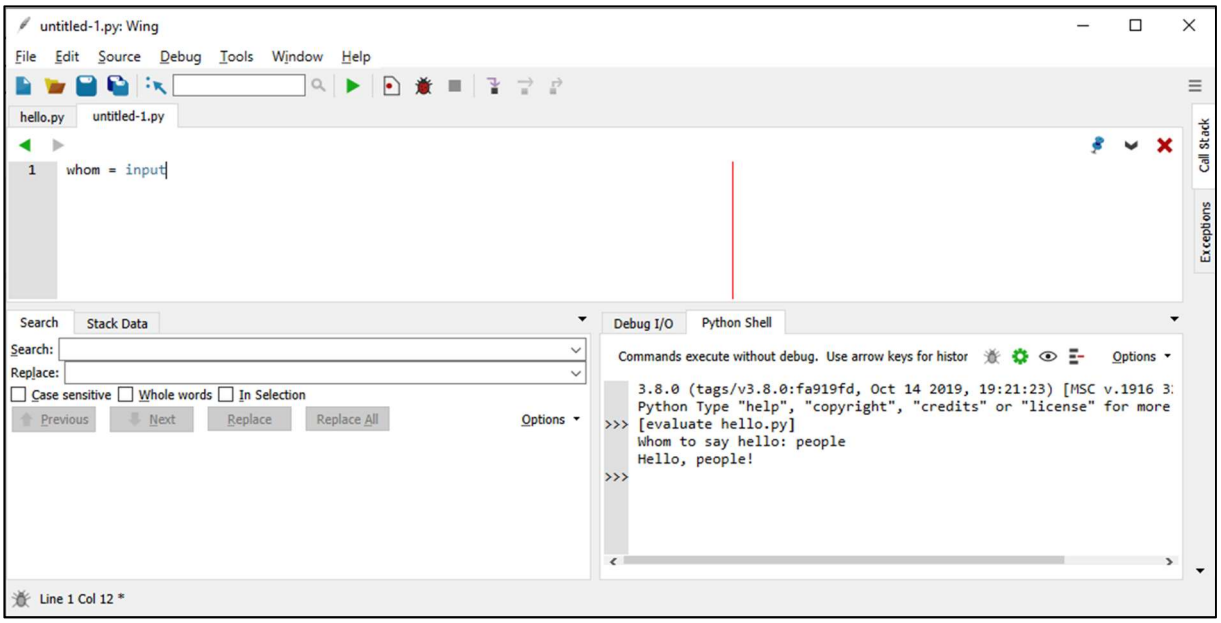
Kad, rakstot tekstu, esam ievadījuši pietiekoši daudz burtus no kāda pazīstama vārda, *Wing* sāk teikt priekšā, kas tas (vai tie) varētu būt par vārdiem:



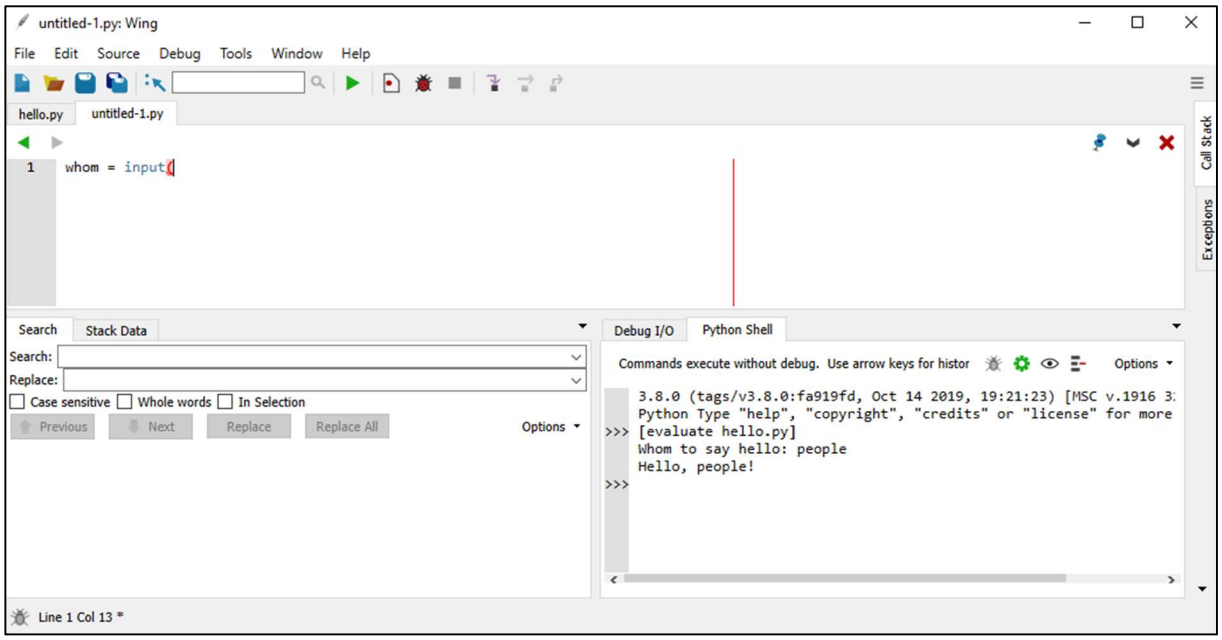
vai



Ja kāds no piedāvātajiem vārdiem der, to var izvēlēties ar bultiņām un izvēlēties ar tabulācijas taustiņu (citās izstrādes vidēs var būt nedaudz citādāk):



Kad pēc funkcijas vārda vēl ievada atverošo iekavu, *Wing* profesionālā versija vēl saka priekšā, ko tur var rakstīt (kā parametru/parametrus), bet *Wing 101* versijā, kas ir par brīvu, šāda lieta nav iekļauta, un tālāk jāraksta pašam...



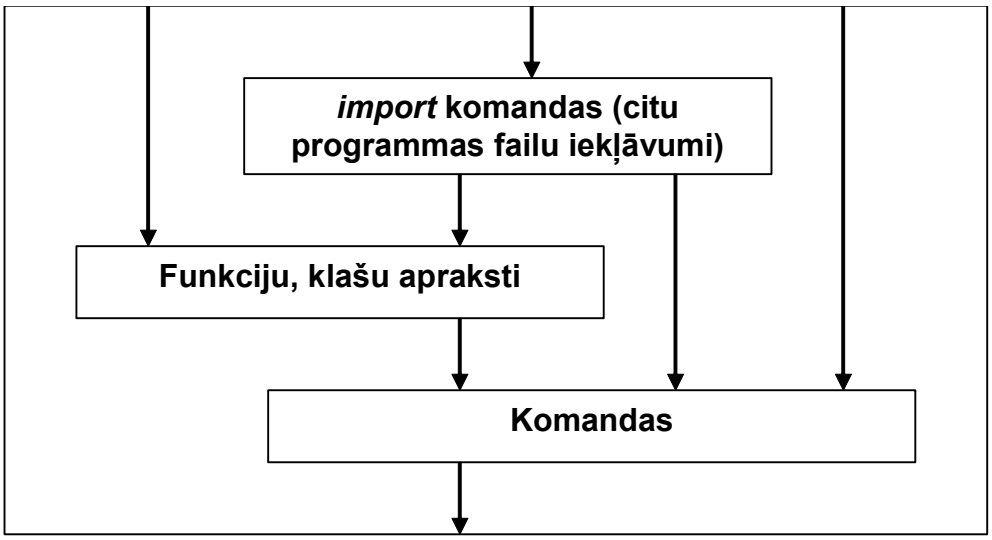
2.5. *Python* programmas vispārējā struktūra

2.5.1. Programmas struktūras vispārējā shēma un darbināšanas principi

Valodas *Python* programma (kā jebkurā citā programmēšanas valodā rakstīta programma) ir konstrukciju kopums, kas apraksta noteiktu algoritmu.

Python programma no valodas konstrukciju viedokļa ir dažādu komandu un nosaukto bloku (funkciju un klašu) kopums (Att. 2.3). Programmas darbināšana sākas no faila sākuma (sk. arī piemēru tālāk Att. 2.4), ejot pa (pirmā hierarhijas līmeņa, t.i., bez atkāpes) elementiem:

1. sastopot komandu, tā tiek izpildīta,
    a. ja šī komanda ir moduļa ielādes komanda *import*, tad tiek izsaukta attiecīgā moduļa (t.i., faila) darbināšana pēc šī paša principa – tātad, ielādējot moduli, tur esošās "parastās" komandas tiek izpildītas, nevis ielādētas,
2. sastopot nosauktu bloku (funkciju vai klasi), tas tiek ielādēts.



Att. 2.3. *Python* programmas vienkāršota struktūra (dotie elementi var būt arī citā secībā)

No leksiskā viedokļa Python programmas struktūru veido **leksēmas** (*lexemes*) jeb **marķieri** (*tokens*):

- identifikatori (*identifiers*),
- atslēgas vārdi (*keywords*),
- literāļi jeb konstantās vērtības (*literals*),
- operatoru marķieri (*operator tokens*) un
- dienesta simboli (*punctuators*).

Sintaktiskā līmenī programmu veido:

- mainīgie (*variables*),
- datu tipi (*data types*),
- funkcijas (*functions*) – funkciju izsaukumi un signatūras,
- izteiksmes (*expressions*).

Konstrukciju līmenī varētu teikt, ka Python programmas sastāvā ietilpst:

- priekšraksti (*statements*),
- bloki (*blocks*),
- bloku galvas.

Gan sintaktiskajā, gan konstrukciju līmenī elementi var būt strukturēti rekursīvi, t.i., ietilpt viens otrā.

2.5.2. Vienkāršas programmas piemērs

Programma parasti sākas ar standarta bibliotēku vai citu moduļu ielādi, kam var sekot funkciju un izpildāmā koda apraksts.

Programmas (Att. 2.4) darbināšana notiek šādi:

- Rindā 1 – tiek ielādēta bibliotēka *sys* (mēs šo bibliotēku izmantosim izmantotās *Python* versijas noskaidrošanai).
- Rindā 3 – tiek iegūta un izdrukāta *Python* versija, ar kuru strādājam (3.8.0).
- Rindās 5,6 – tiek ielādēta (un nevis izpildīta) funkcija *process*.
- Rindās 8,9,11 – tiek definēti un izdrukāti mainīgie x,y (5,7).
- Rindā 13 tiek izsaukta funkcija *process* un izdrukāts, ko tā atgriež (skaitļu 5 un 7 summu 12 un reizinājumu 35).

```
1. import sys
2.
3. print(sys.version)
4.
5. def process(a,b):
6.     return a+b,a*b
7.
8. x=5
9. y=7
10.
11.   print(x,y)
12.
13.   print(process(x,y))
```

```
3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916
                32 bit (Intel)]
5 7
(12, 35)
```

Att. 2.4. Programmas piemērs valodā Python

No leksiskā viedokļa šī programma ietver šādus elementus:

- identifikatori (ieskaitot rezervētos vārdus): sys, print, process, a, b, x, y;
- atslēgas vārdi: import, def, return
- literāļi: [5] [7]
- operatori un dienesta simboli: () + * : = , .

No sintaktiskā viedokļa programma ietver šādus elementus:

- mainīgie: x, y, a, b, sys
- konstantes: visi literāļi
- funkcijas: print process
- objektu elementi: version
- izteiksmes: (piemēram) [a * b] [a + b] process(x,y) (arī mainīgie, literāļi un funkciju izsaukumi uzskatāmi par vienkāršām izteiksmēm)

No konstrukciju viedokļa programmā ir izdalāmas šādas daļas:

- priekšraksti: (piemēram) [x=5] [print(process(x,y))]
- bloki: blakusesošo rindiņu kopumi ar vienādu atkāpi no kreisās
- galvas [def process(a,b):]

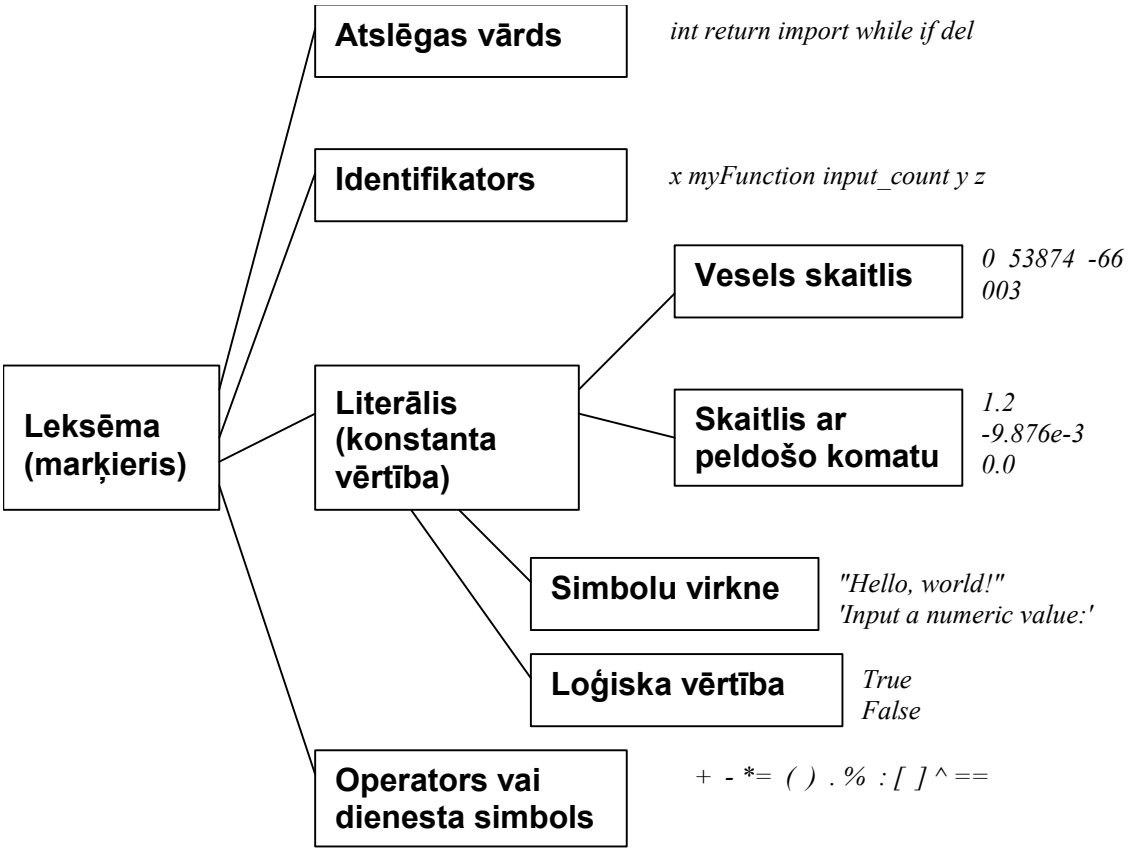
3. Valodas *Python* struktūras elementi

3.1. Python programmas leksiskā līmeņa elementi

No leksiskā jeb zemākā līmeņa pieraksta viedokļa Python programma sastāv no leksēmām jeb marķieriem.

> **Leksēma** (*lexeme*) (programmēšanas valodā) ir mazākā valodas vienība.

Piemēram, cilvēku valodā leksēmas (no programmēšanas valodu viedokļa) ir vārdi un pieturzīmes, bet viens burts kādā vārdā vairs nav leksēma.



Att. 3.1. *Python* leksiskā līmeņa elementu klasifikācija

Praktiski leksēma ir viena rakstzīme vai vairāku pēc kārtas esošu rakstzīmju secība. Savukārt programma ir leksēmu secība.

Ir divi veidi, kā leksēmas var tikt atdalītas viena no otras:

- izmantojot atdalītājsimbolus (tukšums, tabulācija, jaunas rindiņas simbols (ENTER) u.c.),
- pēc konstrukcijas – simbola nepiederība dotajai leksēmai norāda uz nākošās leksēmas sākšanos (piemēram, programmas teksta fragmentā, neskaitot kvadrātiekavas, [**1+2**], ir trīs leksēmas, kaut arī nav atdalītājsimbolu).

Dažu leksēmu atdalīšanai vienai no otras obligāti jālieto atdalītājsimboli.

3.1.1. Identifikatori

> **Identifikators** (*identifier*) ir mainīgā vai cita programmēšanas elementa vārds.

Rezervētajiem vārdiem valodā *Python* ir noteikta nozīme, kuru nedrīkst mainīt, tomēr saviem mainīgajiem un citiem programmas elementiem programmētājs var brīvi izvēlēties identifikatorus, tomēr tie nedrīkst konfliktēt ar rezervētajiem vārdiem un citiem identifikatoriem. Identifikators var sākties ar burtu vai pasvītrojuma zīmi (_), bet pārējie simboli var būt arī cipari. Tipiskākais identifikatora piemērs ir mainīgā vārds.

Python identifikatoru pieraksts (atšķirībā no dažām programmēšanas valodām, kas gan ir mazākumā, kā, piemēram, *Basic*, *Pascal*) ir **reģistrjūtīgs** (*case-sensitive*), t.i., piemēram, identifikatori *count* un *COUNT* ir uzskatāmi par diviem dažādiem.

Korekti identifikatori ir, piemēram, šādi:

```
x sum input_count _x ExchangeRate_1 a2b3c
```

Šādas simbolu virknes nevar kalpot par identifikatoriem:

```
50 2b3c ExchangeRate-1 input.count %x
```

3.1.2. Atslēgas vārdi

> **Atslēgas vārdi** (*keywords*) ir programmēšanas valodā iekšēji definēti vārdi.

Atslēgas vārdus apraksta identifikatori. Atslēgas vārdi apraksta iebūvētos datu tipus, dažādas konstrukcijas un citus valodas elementus. *Python* pieejamie atslēgas vārdi uzskaitīti Tab. 3.1.

Tab. 3.1.

Atslēgas vārdi

```
and as assert break class continue def del elif else except False
finally for from global if import in is lambda None nonlocal not or
pass raise return True try while with yield
```

3.1.3. Literāļi

> **Literāļi** (*literals*) ir dažādu tipu konstantās vērtības, kas tiešā veidā parādās programmas tekstā.

Literāļi var būt veseli skaitļi, skaitļi ar peldošo komatu, simbolu virknes un loģiskās vērtības (sk. arī Att. 3.1).

3.1.3.1. Vesels skaitlis

Vesels skaitlis vienkāršākajā variantā ir pierakstāms kā (decimālu) ciparu virkne, pirms kuras var atrasties plus vai mīnus zīme. (vēl Python pieļauj pierakstu arī binārā, astotnieku un sešpadsmitnieku formā).

Korekti pierakstīti veseli skaitļi ir šādi:

```
1234 0 003 +28 -837
```

Šādas simbolu virknes neapzīmē veselus skaitļus:

```
50.0 3x '1234' "1234" 'abc' abc
```

3.1.3.2. Skaitlis ar peldošo komatu

Skaitlis ar peldošo komatu sastāv no veselās daļas un mantisas (daļas pēc decimālās zīmes), kam var sekot desmitnieku pakāpe, ko ievada simbols ‘e’ vai ‘E’: -1.2e-3 nozīmē $-1.2*10^{-3}$ jeb -0.0012.

Vienu no divām – veselo daļu vai mantisu var izlaist, tādējādi ‘0.3’ var pierakstīt kā ‘.3’ , bet ‘4.0’ kā ‘4.’.

Korekti skaitļi ar peldošo komatu ir:

```
1.0  -0.0  -.4  3.  1.23e4  2.e-3  .5e+2  6E1
```

Šādas virknes nevar kalpot par skaitļiem:

```
1.2.3  3..4  5.e  5.e1.1  .  .e2
```

3.1.3.3. Simbolu virknes literālis

Simbolu virknes literālis (*string literal*) apzīmē virkni no vairākiem simboliem. Virkne drīkst būt arī tukša (bez neviena simbola). To pieraksta, liekot atbilstošās rakstzīmes parastajās vai dubultajās pēdiņās, speciālos simbolus pierakstot tāpat kā simbola literāļa gadījumā (\n nozīmē jaunas rindiņas simbolu, atpakaļsvītra \ vispār nozīmē palīgzīmi, kur virknē liekamais simbols (parasti kaut kāds speciālais, neredzamais simbols) tiek noteikts kopā ar nākošo vai nākošajām pieraksta zīmēm):

```
""  "a"  "Hello, World!"  'Hello'  "Hello,\nWorld!"  "\'Hello\'"
"'Hello'"
```

3.1.3.4. Loģiskās vērtības literālis

Loģiskās vērtības literālis (*Boolean literal*) apzīmē loģisku vērtību ar tipu *bool*:

```
True False
```

3.2. Daži Python programmas sintaktiskā līmeņa elementi

Galvenās valodas *Python* sintaktiskās pamatvienības ir mainīgie, konstantes, datu tipi, kā arī funkcijas un izteiksmes.

3.2.1. Mainīgie

> **Mainīgais** (*variable*) ir mehānisms, kas programmā reprezentē noteiktu vērtību.

Mainīgo programmā identificē noteikts vārds – identifikators. Valodā *Python* darbs ar mainīgo sākas, tam pirmo reizi piešķirot vērtību, t.i., to inicializējot:

```
a = 123
```

> **Inicializācija** (*initialization*) ir mainīgā pirmreizējā aizpildīšana ar vērtību.

Ja vajag mainīgo bez vērtības, tad šīm nolūkam kalpo tukšā vērtība None:

```
a = None
```

Atšķirībā no dažām citām programmēšanas valodām, Python mainīgajiem:

- neeksistē speciāls un nodalīts jēdziens – mainīgā deklarācija (t.i., paziņojums, ka tāds mainīgais tiks tālāk programmā izmantots) – tas notiek automātiski, pirmo reizi mainīgajam piešķirot vērtību (izņēmums ir vienīgi globālo mainīgo izmantošana funkcijās rakstīšanas režīmā),
- mainīgajam nav datu tipa (datu tips ir tikai vērtībai, uz ko norāda mainīgais).

Valodā *Python* mainīgais savā ziņā ir tikai nosaukums, kas apzīmē vērtību.

No redzamības viedokļa dažādos programmas moduļos mainīgos iedala šādās grupās:

- globālie (*global variables*),
- lokālie (*local variables*) – funkciju iekšējie mainīgie,

Par globālajiem un lokālajiem mainīgiem tiks runāts vēlāk, nodaļā par funkcijām.

3.2.2. Operatori

> **Operators** (*operator*) ir programmas konstrukcijas elements, kas nosaka noteiktu darbību ar datiem.

Operatori kopā ar tajos iesaistītajiem datiem un citiem konstrukcijas elementiem veido tādas valodas konstrukcijas kā izteiksmes (aprakstītas nākamajās nodaļās) un priekšraksti (*statements*). No pieraksta viedokļa operators ir viena (parasti) vai vairākas leksēmas, kas var būt izkliedētas vienas izteiksmes vai priekšraksta ietvaros (līdzīgi kā saiklis *gan..., gan* latviešu valodā).

Šaurākā nozīmē ar operatoriem saprot aritmētiskos, piešķiršanas, salīdzināšanas, loģiskos utml. operatorus (tā tas ir aprakstīts arī šeit), bet plašākā nozīmē tas var apzīmēt arī valodas kontroles konstrukciju elementus (piemēram, *if-elif-else*, *while*).

Atkarībā no iesaistīto datu vienība skaita, operatori ir **unāri** vai **bināri**. Lielākā daļa operatoru ir bināri (saistīti ar divām datu vienībām).

Ar operatoriem neatraujami saistīts jēdziens ir prioritāte.

> **Prioritāte** (*priority*) ir operatoru raksturojošs lielums, kas nosaka operatora izpildes vietu vairāku operatoru izpildes secībā.

Piemēram, reizināšanas augstāka prioritāte pār saskaitīšanu nodrošina, ka izteiksme *2+3*4* izpildīsies kā *2+(3*4)*.

Katram operatoram bez prioritātes ir noteikts arī izpildes virziens, kurš iegūst nozīmi gadījumos, kad blakusesošo operatoru prioritātes ir vienādas.

Unārajiem operatoriem, kā arī piešķiršanas operatoram, izpildes virziens ir no labās uz kreiso (piemēram, *x=y=z* nozīmē *x=(y=z)*), bet pārējiem – no kreisās uz labo (piemēram, *x+y+z* nozīmē *(x+y)+z*).

Tab. 3.2.

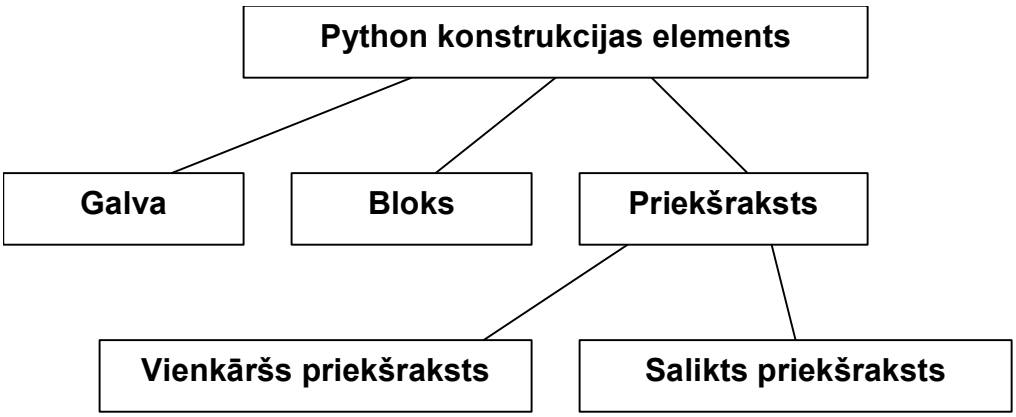
Galvenās Python operatoru grupas

aritmētiskie operatori	`+ - * / % // **`
loģiskie operatori	`and or not`
piešķiršanas operatori	`= += -= *= /= %= //= **=`
salīdzināšanas un identitātes operatori	`== != < <= > >= is (is not)`
piederības operatori	`in (not in)`
bitu līmeņa operatori un	`& | ^ ~ << >>`
bitu līmeņa piešķiršanas operatori	`&= |= ^= ~= <<= >>=`
(Tālāk uzskaitītie arī varētu tikt uzskatīti par operatoriem, kaut arī formāli valodā *Python* neskaitās operatori:)	
piekļuve saraksta, slēgtā saraksta, vārdnīcas elementiem, saraksta (*list*) apzīmētājs	`[]`
vārdnīcas (*dict*) apzīmētājs	`{}`
slēgtā saraksta (*tuple*) apzīmētājs	`()`
vienību atdalītājs	`,`
vārdnīcas, intervāla vienības atdalītājs	`:`
kondicionālais operators	`( if else )`
saraksta atpakošanas operators	`*`
vārdnīcas atpakošanas operators	`**`

3.3. Python programmas konstrukcijas līmeņa elementi

Viena no programmēšanas valodas (t.sk. Python) svarīgākajām konstrukcijām ir priekšraksts (statement). Varētu teikt, ka no konstrukciju viedokļa programma sastāv pamatā no **priekšrakstiem**.

> **Priekšraksts** (*statement*) ir pabeigta valodas konstrukcija, kas veic noteiktu darbību.



Att. 3.2. Valodas Python konstruktīvo elementu klasifikācija

Priekšraksti var tikt apvienoti, veidojot **moduļus** vai **saliktos priekšrakstus**. Tipiska konstrukcija priekšrakstu apvienošanai ir **bloks**.

> **Bloks** (*block*) ir secīgu priekšrakstu virkne, kas atrodas vienā hierarhijas līmenī (*Python* – ar vienāda lieluma atkāpi no kreisās malas).

Priekšrakstu apvienošana blokos var notikt daudzos līmeņos, tādējādi bloks var būt arī kāda priekšraksta sastāvdaļa.

Salikti priekšraksti (*compound statements*) ir tādi priekšraksti, kuru sastāvā ietilpst citi priekšraksti. **Vienkāršu priekšrakstu** (*simple statements*) sastāvā nevar ietilpt citi priekšraksti. Šajā mācību materiālā par saliktajiem priekšrakstiem sauksim tikai tādus, kuru sastāvā var ietilpt bloki, taču kaskādes veidā izsaucamus priekšrakstus (piemēram, piešķiršanas priekšrakstu) pieskaitīsim pie vienkāršajiem. Saliktu priekšrakstu piemēri ir izvēles un cikla priekšraksti, bet vienkāršu priekšrakstu piemēri – piešķiršanas, funkcijas izsaukuma, atdalīšanas priekšraksti.

Priekšrakstus, kuru pieraksts programmas tekstā nosacīti ievietojas vienā rindā (bet dažreiz arī vienkārši programmas teksta daļu, kas ievietojas vienā rindiņā), mēdz saukt par **instrukcijām** (*instruction*). Par instrukcijām kalpo vienkāršie priekšraksti un var teikt, ka programmas pamatmasu veido tieši instrukcijas – tās uzskatāmas par programmas struktūras ķieģeļiem.

4. *Python* pamati

4.1. Vērtības un datu tipi

Programma kādā programmēšanas valodā realizē vienu vai vairākus algoritmus, kas ir darbību vai instrukciju virkne. Tomēr šīs darbības darbojas ar datiem, un programmas darba rezultāts ir tieši dati, respektīvi, kaut kādas vērtības. Parasti programmēšanas valodās dati nepastāv kā vērtības vien. Neatņemama vērtības sastāvdaļa ir datu tips. Piemēri datu tipiem ir vesels skaitlis, skaitlis ar peldošo komatu, teksts.

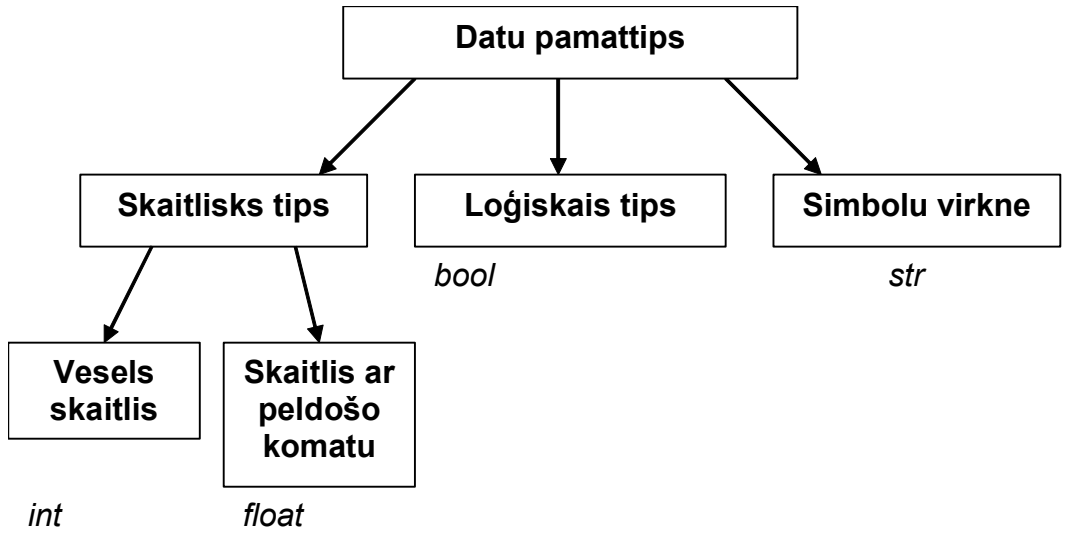
> **Datu tips** (*data type*) (no vispārēja izmantošanas viedokļa) ir datu veids.

Tomēr programmēšanā datu tips ir kas vairāk nekā tikai datu veids.

> **Datu tips** (*data type*) (no datu apstrādes viedokļa) ir darbību kopums, ko var veikt ar noteiktu vērtību.

Tādējādi it kā viena un tā pati darbība ar dažādu datu tipu vērtībām var tikt veikta dažādi. Piemēram operācija '+' skaitliskajiem datu tipiem nozīmē saskaitīšanu, bet simbolu virknēm - konkatenāciju.

Valodas *Python* vienkāršo pamattipu klasifikācija parādīta Att. 4.1.



Att. 4.1. Valodas *Python* vienkāršie datu pamattipi

No vienkāršajiem datu pamattipiem var būvēt saliktus datu tipus – sarakstus, klases (sk. tālākajās nodaļās).

Katru datu tipu raksturo:

- atmiņas daudzums (veselos baitos), ko aizņem viena šāda tipa vērtība,
- vērtību diapazons, kas atbilst šim datu tipam,
- ar šo datu tipu saistītās darbības.

Nākošajā tabulā parādīti *Python* vienkāršie pamattipi un dots īss to raksturojums.

Tab. 4.1.

Python vienkāršie pamattipi

Datu tips	Tipiski bāzes izmēri baitos	Tipiski bāzes izmēri baitos Python 3.8.0, kas izmantots materiālā*	Vērtību diapazons	Piemēri
int (vesels skaitlis)	mainīgs, neierobežots, pēc vajadzības	12 un vairāk	neierobežots	`-8 4567 +20478 0 123546789123456789`
float (skaitlis ar peldošo komatu)	8	8 + tehniskie baiti	$\pm\sim10^{-308}..10^{308}$ (ja 8B)	`1.286 -1.2e+9`
str (simbolu virkne)	1..N katram simbolam, kur N var būt pat divciparu skaitlis	1..N katram simbolam, kur N var būt pat divciparu skaitlis + tehniskie baiti	katrs simbols: jebkurš Unicode simbols	`'Alpha'` `"hello world"`
bool (logiskais)	tāpat kā **int**	12 (False), 14 (True)	True (1), False (0)	`True False`

* papildus bāzes vietai vērtību glabāšanā tiek izmantota papildus atmiņa, un valodā Python papildu atmiņa tiek izmantota salīdzinoši daudz

Ērtību nodrošināšanas dēļ *Python* ir salīdzinoši izšķērdīgs pret atmiņu datu glabāšanai tā pamattipos, tāpēc lielu datu efektīvai apstrādei tiek izmantotas specializētas bibliotēkas ar optimizētiem datu tipiem, piemēram, *NumPy* bibliotēkas datu tipi *int32*, *float32* utml.

Vēl viens iedalījums – visi *Python* datu tipi iedalās divās grupās, par kurām tiks stāstīts tālākajās nodaļās (par saliktajiem datu pamattipiem un funkcijām):

- slēgtie jeb nemaināmie (*immutable*) – visi nosauktie vienkāršie *Python* pamattipi – *int*, *float*, *str*, *bool* – ir šādi,
- atvērtie jeb maināmie (*mutable*).

Kā jau minēts, valodā *Python* pašam mainīgajam nepiemīt datu tips, tas piemīt tikai vērtībai, uz ko norāda mainīgais.

Galvenie veidi, kā piekārtot vērtībai datu tipu (piemērus sk. zemāk):

- vērtība aprakstīta, izmantojot kādu no literāļiem, kas automātiski nosaka datu tipu (Tab. 4.1) vai konteksta;
- veicot datu tipa pārveidojumu ar funkciju (komandu), kuras nosaukums atbilst datu tipam.

4.1.1. Datu tipa iegūšana no literāļa vai konteksta

Ciparu virkne norāda uz *int* datu tipu, decimālais punkts vai E (pakāpe) – uz *float*, pēdiņas uz tekstu, *True*, *False* vai salīdzināšana – uz *bool*. Funkcija *type* atgriež vērtības datu tipu (programmas vai to fragmenti tiek marķēti ar biezu svītru programmas koda kreisajā pusē, pēc

tam var sekot programmas darbības piemērs, kurš marķēts ar trīskāršo svītru kreisajā pusē, ievads no konsoles iezīmēts treknināti). Tālākajos piemēros tiks izmantota arī funkcija *type*, kas atgriež dotās vērtības tipu.

```
print(type(1))
print(type(1.1))
print(type(1.1e3))
print(type('alpha'))
print(type("beta"))
print(type(True))
```

```
<class 'int'>
<class 'float'>
<class 'float'>
<class 'str'>
<class 'str'>
<class 'bool'>
```

Att. 4-2. Datu tipa paņemšana no literāļa

```
a=123
b=a*0.1
print(type(b))
print(type(2<3))
c=input()
print(type(c))
```

```
<class 'float'>
<class 'bool'>
hello
<class 'str'>
```

Att. 4-3. Datu tipa paņemšana no konteksta

4.1.2. Datu tipa tieša pārveidošana

Datu tipa pārveidošanai izmanto funkcijas, kas pēc nosaukuma sakrīt ar datu tipu – *int*, *float*, *bool*, *str*, piemēram, komanda *str(a)* pārveido vērtību (mainīgajā) *a* uz uz *str* datu tipu.

```
a = 5
b = 7
print('skaitlis',a+b)
print('teksts',str(a)+str(b))
a = str(a)
b = str(b)
print('teksts',a+b)
```

```
skaitlis 12
teksts 57
teksts 57
```

Att. 4-4. int pārveidošana par str (a un b kā skaitļi tiek saskaitīti, a un b kā teksts tiek konkatenēti)

```
a = 5
print(a,type(a))
a = float(a)
print(a,type(a))
a = int(a)
print(a,type(a))
a = 123.56
a = int(a)
print(a,type(a))
```

```
5 <class 'int'>
5.0 <class 'float'>
5 <class 'int'>
123 <class 'int'>
```

Att. 4-5. int pārveidošana par float un atpakaļ

```
s = "123"
t = "456"
print(s+t, type(s+t))
a = int(s)
b = int(t)
print(a+b, type(a+b))
```

```
123456 <class 'str'>
579 <class 'int'>
```

Att. 4-6. str pārveidošana par int veiksmīgi

```
s = int("abc")
```

```
Traceback (most recent call last):
  File "C:/src/datatypes.py", line 1, in <module>
    s = int("abc")
builtins.ValueError: invalid literal for int() with base 10:
                     'abc'
```

Att. 4-7. str pārveidošana par int neveiksmīgi

4.2. Piešķiršana

> **Piešķiršana** (*assignment*) ir mainīgā uzstādīšana norādīšanai uz noteiktu vērtību.

Piešķiršanu veic **piešķiršanas priekšraksts** (*assignment statement*), kas ietver sevī piešķiršanas operatoru (vienlīdzības zīmi).

Labajā puse var būt jebkura konstrukcija, kas reprezentē vērtību (piemēram, literālis, mainīgais, funkcijas izsaukums vai izteiksme).

```
x = 5
x = y = z
x = (y + z) * 5
```

Att. 4.8. Piešķiršanas priekšraksta piemēri

Kā redzams piemērā, piešķiršanas operatori ir izmantojami viens pēc otra kaskādes veidā.

Papildus operatoram =, ir pieejami papildus piešķiršanas operatori komplektā ar aritmētisku operāciju (sk. Tab. 3.2), piemēram,

```
x += 2
```

ir tas pats, kas

```
x = x + 2
```

4.3. Aritmētiskas izteiksmes un galvenās skaitliskās funkcijas

> **Izteiksme** (*expression*) ir konstrukcija, kas sastāv no operatoriem (jeb operācijām) un operandiem (parasti vērtībām), kas nosaka rēķināšanas procesu.

Parasti rēķināšanas procesa rezultāts ir noteikta tipa vērtība.

Vienkāršākais izteiksmju variants ir aritmētiskas izteiksmes, kuras atšķirībā no matemātikas pieraksta lineāri, izpildes secību ietekmējot ar papildus iekavām.

Piemēram, aritmētisku izteiksmi

$$\frac{\sin(x)+1}{y}+\frac{y}{x+1.5}$$

valodā *Python* pieraksta:

```
(math.sin(x) + 1) / y + y / (x + 1.5)
```

Vispārējā programmēšanas teorijā parasti tiek izšķirtas aritmētiskas izteiksmes un loģiskas izteiksmes, tomēr mūsdienu programmēšanas valodās (arī *Python*) izteiksme vispārīgā gadījumā ne tikai apvieno abus šos izteiksmju veidus, bet arī saturīgi krietni pārsniedz šo abu izteiksmju veidu apvienojumu. Neskatoties uz to, šajā mācību materiālā, lai papildus nodrošinātu lasītāju izpratni, gan aritmētiskās, gan loģiskās izteiksmes tiks apskatītas atsevišķi, lai uzskatāmāk parādītu to tipisko pielietojumu dažādās konstrukcijās.

> **Aritmētiska izteiksme** (*numeric expression*) ir konstrukcija, kas sastāv no skaitliskām operācijām un skaitliskiem operandiem, un kura pēc šo operāciju izpildes izrēķina skaitlisku vērtību.

Aritmētisku izteiksmju "motors" ir aritmētiskas operācijas un funkcijas.

Galvenās *Python* skaitliskās operācijas (*numeric operators*) parādītas Tab. 3.2.

Savukārt operandi var būt skaitliskas vērtības (literāļi), mainīgie un funkcijas.

Tālāk aprakstītas dažas skaitliskas funkcijas (dažas no tām pieejamas bibliotēkā *math*, kas speciāli jānorāda). Ja tiek izmantotas funkcijas no bibliotēkas *math*, programmas sākumā bibliotēka jāielādē vienā no diviem veidiem:

```
import math
```

vai

```
from math import *
```

Otrajā gadījumā, izsaucot funkciju, tai nebūs priekšā jāliek "*math.*".

pow

```
pow (base,p);
```

Funkcija *pow()* atgriež vērtību, ko iegūst, argumentu *base* kāpinot pakāpē *p*, tas pats, kas *base**p*.

ceil

```
math.ceil (num);
```

Funkcija *ceil()* atgriež mazāko veselo skaitli, kas lielāks vai vienāds par argumentu *num* (noapaļošana uz augšu).

floor

```
math.floor (num);
```

Funkcija *floor()* atgriež lielāko veselo skaitli, kas mazāks vai vienāds par argumentu *num* (noapaļošana uz apakšu).

round

```
round (num);
```

Funkcija *round()* atgriež noapaļotu argumenta *num* vērtību uz tuvāko veselo skaitli.

sin, cos, tan, asin, acos, atan, sinh, cosh, tanh

```
math.sin (arg);
```

Funkcijas *sin()* u.c. realizē attiecīgās trigonometriskās funkcijas vai to hiperboliskos variantus. Argumenta vērtība tiek mērīta radiānos.

exp

```
math.exp (arg);
```

Funkcija *exp()* atgriež argumenta *arg* eksponenti (skaitļa *e* kāpinājumu pakāpē *arg*).

log

```
math.log (arg);
```

Funkcija *log()* atgriež argumenta *arg* naturālo logaritmu.

log10

```
math.log10 (arg);
```

Funkcija *log10()* atgriež argumenta *arg* logaritmu pie bāzes 10.

log2

```
math.log2 (arg);
```

Funkcija *log2()* atgriež argumenta *arg* logaritmu pie bāzes 2.

sqrt

```
math.sqrt (arg);
```

Funkcija *sqrt()* atgriež argumenta *arg* kvadrātsakni, tas pats, kas *arg**0.5*.

abs

```
abs (arg);
```

Funkcija *fabs()* atgriež argumenta *arg* absolūto (t.i., bez zīmes) vērtību.

```
import math
print(abs(-55))
print(round(6.8))
print(math.log2(8))

from math import *
print(log2(8))
```

```
55
7
3.0
3.0
```

Att. 4-9. Matemātisko funkciju izmantošanas piemēri

5. Standarta ievade un izvade

Lai programma darbotos, tai vispirms ir jāpadod noteikti dati, bet programmas darbības beigās mums būtu jāsaņem rezultāts. Ar to nodarbojas ievade un izvade. Iepriekš apskatītajos programmu piemēros jau bija redzamas izvades un ievades funkcijas print un input.

Ievade valodā *Python* notiek tikai teksta formā (t.i., lai iegūtu, piemēram, skaitli, ir jāveic tieša pārveidošana no teksta uz skaitli), bet izvades gadījumā *Python* piedāvā automātisku vai netiešu pārveidošanu un formatēšanu.

5.1. Formatēta izvade

5.1.1. Darbības princips

Izvade (uz ekrāna) notiek, izmantojot funkciju *print*.

Lai uz ekrāna izdrukātu skaitli 5, jāraksta

```
print(5)
```

```
5
```

(Izdrukas beigās tiek speciāli tieši parādīts, ka izdrukāts jaunas rindiņas simbols, t.i., notikusi pāreja jaunā rindā.)

Vairākas vērtības iespējams izdrukāt, vienkārši atdalot tās ar komatiem (izdrukā automātiski starp vērtībām liek tukšumus):

```
print(5,6,7,"ok")
```

```
5 6 7 ok
```

Izmantojot nosaukto parametru sep, var uzstādīt citu atdalītāju, šeit – semikols un tukšums:

```
print(5,6,7,"ok", sep="; ")
```

```
5; 6; 7; ok
```

Funkcija print beigās automātiski izdrukā jaunas rindiņas simbolu:

```
print(5)
print(6)
print(7);
```

```
5
6
7
```

To var izmainīt ar parametru end:

```
print(5,end="")
print(6,end=", ")
print(7)
```

```
56, 7
```

5.1.2. Izdrukājamās informācijas formatēšana

Parasti formatēšanu veic skaitļiem ar peldošu komatu:

```
d=12.3456789
print(d)
```

```
12.3456789
```

Formatēšanai izmanto papildus metodi *format*, kas tiek piemērota formatējumam, kur figūriekavu pāri nozīmē vietas, kur ievietot datus, bet paši dati padoti kā parametri:

```
d=12.3456789
print("{} * {}".format(d,33))
```

```
12.3456789 * 33
```

Datu pozīcijas var mainīt, ievietojot numurus figūriekavās (parametros datu pozīcijas numurējas 0, 1, ...):

```
d=12.3456789
print("{1} * {0}".format(d,33))
```

```
33 * 12.3456789
```

Izmantojot papildus modifikatoru *f*, var noteikt ciparu skaitu aiz komata, šeit 3:

```
d=12.3456789
print("{0:.3f}".format(d))
```

```
12.346
```

Izmantojot papildus modifikatoru *e*, ciparu skaits tiek noteikts zinātniski, t.i., tiek ņemti zīmīgie cipari:

```
d=12.3456789
print("{0:.3e} {0:.5e}".format(d,d))
```

```
1.235e+01 1.23457e+01
```

Teksta formatēšanā var izmantot, ka operators *n nozīmē teksta atkārtošanu n reizes konkatenējot:

```
print("#"*10)
```

```
##########
```

Teksta izdrukāšana fiksētā apgabalā ar aizpildīšanu:

```
x="alpha"
y="beta"
z="gamma"
print(x,y,z)
areasize=10
```

```
print('_'*(areasize-len(x)),x,sep="",end="")
print('_'*(areasize-len(y)),y,sep="",end="")
print('_'*(areasize-len(z)),z,sep="",end="")
print()
print(x,'_'*(areasize-len(x)),sep="",end="")
print(y,'_'*(areasize-len(x)),sep="",end="")
print(z,'_'*(areasize-len(x)),sep="",end="")
```

```
alpha beta gamma
_____alpha______beta_____gamma
alpha_____beta_____gamma_____
```

5.2. Ievade

Ievadi no konsoles veic ar funkciju *input()*:

```
s = input()
print(s)
```

```
people
people
```

Funkcijai *input()* var padot parametru, kas nozīmē uzaicinošo tekstu (*prompt*):

```
s = input("Ievadi kaut ko: ")
print(s)
```

```
Ievadi kaut ko: people
people
```

Lai ievadītu skaitli, tas pēc tam tiešā veidā jāpārveido no teksta uz skaitli:

```
s = input("Ievadi skaitli: ")
print(s+"456")
t = int(s)
print(t+456)
```

```
Ievadi skaitli: 123
123456
579
```

Ievadot tekstu, kas nav pārveidojams uz skaitli, tiek konstatēta kļūda un "izmests" izņēmums *ValueError*:

```
s = input("Ievadi skaitli: ")
t = int(s)
```

```
Ievadi skaitli: abc
Traceback (most recent call last):
  File "C:/src/input.py", line 2, in <module>
    t = int(s)
builtins.ValueError: invalid literal for int() with base 10:
                  'abc'
```

Nekorektu ievadu var apstrādāt, izmantojot izņēmumu apstrādes mehānismu *try-except* (ievērojiet, ka rindiņas "zem" *try* un *except* ir vienu atkāpi pa labi);

```
s = input("Ievadi skaitli: ")
try:
    num = int(s)
    print(num,"ok")
except ValueError:
    print("Input error: [{}]".format(s))
```

```
Ievadi skaitli: abc
Input error: [abc]
```

Normālā situācijā viss notiek, kā aprakstīts *try* blokā:

```
s = input("Ievadi skaitli: ")
try:
    num = int(s)
    print(num,"ok")
except ValueError:
    print("Input error: [{}]".format(s))
```

```
Ievadi skaitli: 123
123 ok
```

Ja vienā ievadā ir vairākas vērtības, tad tiek izmantotas dažādas teksta apstrādes funkcijas un citas konstrukcijas, lai apstrādātu tās visas:

```
s = input("Ievadi skaitļus: ")
print(s)
ss = s.split()
print(ss)
summa = 0  # summa no ievadītajiem skaitļiem
for n in ss:
    summa += int(n)
print("Summa:",summa)
```

```
Ievadi skaitļus: 1 2 3 4 5
1 2 3 4 5
['1', '2', '3', '4', '5']
Summa: 15
```

6. Zarošanās un loģiskās izteiksmes

Zarošanās (*branching*, *selection*) ir viena no 3 galvenajām vadības konstrukcijām (blakus secībai (*sequence*) un ciklam (*looping*). Zarošanos valodā Python realizē zarošanās priekšraksts (*selection statements*) *if-elif-else*. Kaut kādā nozīmē ar izvēli nodarbojas arī kondicionālais operators (*if else*), tomēr tas nenosaka izmaiņas programmas vadībā, bet tikai nodrošina noteiktas vērtības izvēli un ir uzskatāms par cita līmeņa konstrukciju.

6.1. Python programmas strukturēšana blokos

Runājot par zarošanās priekšrakstu valodā *Python*, pirmoreiz parādās nepieciešamība pierakstīt vairāku līmeņu hierarhisku struktūru. Valodā Python visa programma strukturēta hierarhiskos blokos, kur bloka līmeni raksturo atkāpju skaits no kreisās malas, un attiecīgais bloks ir pakārtots rindiņai tieši pirms bloka un ar kolu beigās, kurai ir par vienu atkāpi mazāk (Att. 6-1).

```
1. pirmais līmenis
2. pirmais līmenis ar sekojošu pakārtotu bloku:
3.     otrais līmenis (pakārtots rindiņai 2)
4.     otrais līmenis (pakārtots rindiņai 2) ar sek. pak. bl.:
5.         trešais līmenis (pakārtots rindiņai 4)
6.         trešais līmenis (pakārtots rindiņai 4)
7.     otrais līmenis (pakārtots rindiņai 2)
8.     otrais līmenis (pakārtots rindiņai 2)
9. pirmais līmenis
```

Att. 6-1. Shematisks *Python* programmas izkārtojums dažādu līmeņu blokos

Atkāpe var būt gan viens tukšums, gan vairāki tukšumi, gan tabulācijas simbols – galvenais, lai būtu konsekventi. Parasti *Python* teksta redaktori nosaka atkāpi kā 4 tukšumus.

Ja pakārtotais bloks ir vienas rindiņas garumā, tad to drīkst rakstīt tajā pašā rindiņā uzreiz aiz kola, tādējādi ietaupot vienu rindiņu programmas kodā (sk. ceturto rindiņu Att. 6-2)

```
1. pirmais līmenis
2. pirmais līmenis ar sekojošu pakārtotu bloku nākošajā r.:
3.     otrais līmenis (pakārtots rindiņai 2)
4. pirmais līmenis: <pakārtotais bloks tepat>
```

Att. 6-2. Shematisks *Python* programmas izkārtojums, izmantojot vienas rindiņas pakārtotos blokus

6.2. Zarošanās priekšraksts if-elif-else

Programmas darbības gaitā var rasties nepieciešamība atkarībā no kāda nosacījuma izpildīt vai neizpildīt kādu programmas daļu. To parasti veic ar zarošanās priekšrakstu *if-elif-else*.

```
if <nosacījums 1>:
    <bloks 1>
elif <nosacījums 2>:
    <bloks 2>
elif <nosacījums 3>:
    <bloks 3>
…
else:
    <bloks N>
```

Att. 6.3. Priekšraksta *if-elif-else* shēma (vienīgais obligātais if nosacījums 1 un bloks 1)

Valodas Python *if-elif-else* priekšraksts izpildi sāk ar nosacījumu 1 un tālāk seko šādas darbības un principi:

- neizpildoties nosacījumam, pārlec uz nākošo nosacījumu,
- ja nosacījums izpildās, tad izpilda attiecīgo bloku un izlec no priekšraksta (tātad, nevar izpildīt vairāk par vienu bloku),
- ja neizpildās neviens nosacījums, izpildās bloks N (*else* bloks),
- ja neizpildās neviens nosacījums un *else* bloka nav, neizpildās nekas.

if-elif-else pielietojumu demonstrē nākošais piemērs:

```
if a > 0: print('positive')
else: print('negative')
```

if-elif-else pielietojums ar *elif* komponenti redzams nākošajā piemērā (== nozīmē salīdzināšanu uz vienādību, nejaukt ar =, kas nozīmē piešķiršanu!):

```
if a > 0: print('positive')
elif a == 0: print('zero')
else: print('negative')if (a > 0) cout << "positive" << endl;
```

If-elif-else priekšraksta blokos var atrasties citi priekšraksti, t.sk., citi *if-elif-else* priekšraksti. Nākošais programmas kods vispār kaut ko izdrukā tikai tad, ja *a* ir pāra skaitlis. (*a%2 == 0* nozīmē: dalot *a* ar 2, atlikums sanāk 0):

```
if a%2==0:
    if a > 0: print('positive even')
    else: print('negative even')
```

Testa piemēri, izmantojot iepriekšējo koda fragmentu – ievērojiet, ka divos pēdējos piemēros netiek izdrukāts nekas:

```
a = int(input())
if a%2==0:
    if a > 0: print('positive even')
    else: print('negative even')
```

```
8
positive even
```

```
-8
negative even
```

```
7
-7
```

6.3. Loģiskas izteiksmes

6.3.1. Vispārīgs apraksts

Priekšraksta *if-elif-else* zarošanos nosaka zarošanās nosacījums, kuru reprezentē loģiska izteiksme, kas vienkāršākajā un tipiskākajā gadījumā ir salīdzināšanas operācija, piemēram, *a>0*, tomēr vispārīgā gadījumā tā var būt krietni sarežģītāka.

> **Loģiska izteiksme** (*logical expression*) ir konstrukcija, kas sastāv no loģiskām operācijām un loģiskiem operandiem, un kura pēc šo operāciju izpildes izrēķina loģisku vērtību.

Loģiska izteiksme ir zarošanās priekšraksta *if-elif-else*, kā arī visu cikla priekšrakstu obligāta sastāvdaļa.

Vispārīgā gadījumā loģiska izteiksme dod atbildi *"patiess"*/*"nepatiess"* (*True*/*False*) uz "jautājumu", kas pierakstīts ar loģisko operāciju un operandu palīdzību.

6.3.2. Salīdzināšanas operācijas – vienkāršākās loģiskās konstrukcijas

Tipiskākais vienkāršas loģiskas izteiksmes piemērs, kā jau tika minēts, ir salīdzināšana. Nākošā tabula parāda salīdzināšanas operatorus un to nozīmi.

Tab. 6.1.

Salīdzināšanas operatori

Matemātiskais simbols	Būtība	Python apzīmējums	Piemērs	Matemātiskais ekvivalents
=	*vienāds*	`==`	`x + 7 == 2 * y`	x+7=2y
≠	*nav vienāds*	`!=`	`s != 'n'`	s ≠ 'n'
<	*mazāks*	`<`	`a < b - 5`	a < b-5
≤	*mazāks vai vienāds*	`<=`	`a <= max`	a ≤ max
>	*lielāks*	`>`	`a > b + 5`	a > b+5
≥	*lielāks vai vienāds*	`>=`	`a >= min`	a ≥ min

Jebkura salīdzināšanas operācija, kā jau loģiskai konstrukcijai pienākas, atgriež vērtību *True* vai *False*, un nevienu citu; tehniski ņemot, *True* ir 1, bet *False* ir 0, bet skatoties pretējā virzienā, 0 tiek interpretēta kā *False*, bet jebkurš cits (vesels) skaitlis – kā *True*.

6.3.3. Loģiskie operatori

Dažādas loģiskās vērtības, piemēram, salīdzināšanas operācijas var apvienot, veidojot sarežģītākas loģiskās izteiksmes. Šim nolūkam der loģiskie operatori `not and or`. Loģisko operatoru operandi var būt jebkādas loģiskas vērtības, t.sk., salīdzināšanas operācijas,

funkcijas, t.i., citas loģiskas izteiksmes. Loģisko operatoru aprakstā *True/False* vietā vai paralēli bieži tiks lietoti *1/0*, jo dažkārt tā ir uzskatamāk.

6.3.3.1. Operators and

Operators *and* (loģiskais "un") atgriež *true* tikai tajā gadījumā, ja abi operandi atgriež *true*, citādi atgriež *false*:

Tab. 6.2.

Loģiskais "un" (and)

A	B	A and B
false (0)	false (0)	false (0)
false (0)	true (1)	false (0)
true (1)	false (0)	false (0)
true (1)	true (1)	true (1)

Nākošā loģiskā izteiksme pasaka, **vai** dotais skaitlis **ir** pozitīvs viencipara skaitlis, vai nav.

```
a = int(input())
print (a >= 0 and a <= 9)
```

```
-8
False
```

```
5
True
```

```
99
False
```

Skaitliski *True* ir 1, bet *False* ir 0.

```
a = int(input())
print (a >= 0 and a <= 9)
```

```
-8
0
```

```
5
1
```

```
99
0
```

6.3.3.2. Operators or

Operators *or* (loģiskais "vai") atgriež *true* tajā gadījumā, ja kaut vai viens (t.sk. abi) operandi atgriež *true*, citādi (ja abi operandi ir *false*) atgriež *false*:

Tab. 6.3.

Loģiskais "vai" (or)

A	B	A or B
false (0)	false (0)	false (0)
false (0)	true (1)	true (1)
true (1)	false (0)	true (1)
true (1)	true (1)	true (1)

Nākošā loģiskā izteiksme pasaka, **vai taisnība, ka** dotais skaitlis **nav** pozitīvs viencipara skaitlis.

```
a = int(input())
print (a < 0 or a > 9)
```

```
-8
1
```

```
5
0
```

```
99
1
```

6.3.3.3. Operators not

Operators *!* (noliegums) atgriež *true*, ja tā operands atgriež *false*, un *false*, ja operands atgriež *true*. Parasti nolieguma operators tiek izmantots komplektā ar kādu no abiem pārējiem loģiskajiem operatoriem, jo citādi no tā var izvairīties, izvēloties attiecīgu salīdzināšanas operatoru.

Tab. 6.4.

Noliegums (not)

A	not A
false (0)	true (1)
true (1)	false (0)

Nākošā loģiskā izteiksme pasaka, **vai** dotais skaitlis **ir** pozitīvs viencipara skaitlis, vai nav (salīdzināt ar piemēriem pie operatoriem and un or, no kuriem pirmais dara to pašu, bet otrais pretēji).

```
a = int(input())
print (not(a < 0 or a > 9))
```

```
-8
0
```

```
5
1
```

```
99
0
```

6.3.3.4. Loģisko operatoru prioritātes

Loģisko operatoru prioritātes rinda ir `not and or`, kur stiprākais ir kreisajā pusē. Lai nomainītu izpildes secību, ja tā neatbilst prioritāšu dēļ, tad jālieto papildus iekavas.

Nākošais piemērs pasaka, vai mēneša *m* (1..12) datums *d* (1..31) ir īsā (30 dienu) mēneša pēdējā diena. Papildus iekavas tiek lietotas, jo loģiskais "vai" ir vājāks par loģisko "un".

```
m = int(input())
d = int(input())
print((m==4 or m==6 or m==9 or m==11) and d==30)
```

```
4
30
True
```

```
5
30
False
```

```
9
5
False
```

6.3.4. Aritmētisku un loģisku izteiksmju saistība

Pēc būtības:

- *true* ir 1, bet *false* ir 0,
- pretējā virzienā – 0 ir *false*, bet visas pārējās vērtības ir *true*.

```
print (7>5, 7<5, int(7>5), int(7<5))
a = 5 and False
print(bool(0), bool(1), bool(-3), a)
```

```
True False 1 0
False True True False
```

6.4. Kondicionālais operators (if else):

Kondicionālais operators darbojas pēc līdzīga principa kā *if-elif-else* priekšraksts ar atšķirību, ka veicamo darbību (priekšrakstu) vietā ir izteiksmes, kas atgriež vērtību, tādējādi arī pats operators atgriež noteiktu vērtību – to, kuru izrēķina attiecīgā izteiksme.

Kondicionālā operatora priekšrocība ir ērta izmantošana, veidojot izteiksmes.

```
a = int(input())
print("positive" if a>0 else "not positive")
```

```
5
positive
```

```
-5
not positive
```

6.5. Izteiksmes vispārīgā nozīmē

Iepriekšējās sadaļās tika apskatītas aritmētiskas izteiksmes un loģiskas izteiksmes, uzsverot katras specifisko funkcionalitāti, tai pat laikā vienkāršojot to definēšanu. Tai pat laikā jau sadaļā 6.3.4. "Aritmētisku un loģisku izteiksmju saistība" tika parādīts, ka būtībā aritmētiskas un loģiskas izteiksmes netiek izšķirtas. Patiesībā ir vēl "sliktāk" – programmēšanas valodās izteiksme parasti ir vēl plašāks jēdziens nekā tikai aritmētisku un loģisku izteiksmju apvienojums.

> **Izteiksme** (*expression*) ir konstrukcija, kas sastāv no operatoriem un operandiem un nosaka skaitļošanas procesu.

Operatori izteiksmē var būt jebkādi un arī operandi (noteiktas vērtības reprezentējoši elementi) var būt dažādu tipu – ka tikai operatori tos spēj apstrādāt. Tāpēc vienīgais, kas neļauj izteiksmei būt tikpat plašam jēdzienam kā algoritms vai programma, ir tas, ka operandu sasaistei tiek lietoti **operatori** – un nekādas citas konstrukcijas.

Nākošajā piemērā parādīta izteiksme, kura nav ne aritmētiska, ne loģiska – bet vienkārši izteiksme. Šī izteiksme izdrukā loģisku vērtību (1 vai 0), bet tās operandi ir skaitliskas un teksta vērtības.

Sekojošais piemērs satur izteiksmi

```
a > len(s + "World!") * 2
```

Šī izteiksme atgriež loģisku vērtību, pasakot, vai taisnība, ka skaitlis a, kas ievadīts no klaviatūras, ir lielāks par teksta "Hello, World!" (pieņemot, ka s ir "Hello, ") garumu, kas pareizināts ar 2 (t.i. 26). Vienā un tajā pašā izteiksmē ir teksta darbība (konkatenācija), aritmētiska darbība (saskaitīšana) un loģiska operācija (salīdzināšana).

```
a = int(input())
s = "Hello, "
print(a > len(s + "World!") * 2)
```

```
20
False
```

```
30
True
```

7. Cikla konstrukcijas

> **Cikls** (*looping*) ir vadības konstrukcija, kas atkārto noteiktas darbības izpildi līdz brīdim, kad pārstāj izpildīties noteikts nosacījums.

Programmēšanā mēdz izšķirt divu veidu ciklus:

- cikls ar skaitītāju,
- cikls ar nosacījumu (*conditional looping*).

Ciklā ar skaitītāju pirms cikla izpildes jau ir zināms atkārtošanas reižu skaits, un tas uzskatāms par cikla ar nosacījumu speciālgadījumu.

Divas obligātas jebkura cikla priekšraksta komponentes (atbilstoši cikla definīcijai) ir **cikla nosacījums** un **cikla ķermenis**.

Cikla ķermenis ir programmas bloks cikla konstrukcijas ietvaros, kas nosaka darbību, kas ciklā tiks atkārtota. Viena cikla ķermeņa izpildīšanu vienu reizi sauc par **iterāciju**. Vispārīgā gadījumā cikla izpildē var būt neviena, viena vai vairākas iterācijas.

7.1. Cikls for

Cikls *for* ir cikls ar skaitītāju, kas nozīmē, ka tas darbojas vai nu uz kāda veselu skaitļu intervāla vai kāda (pārskaitāma) elementu kopuma elementiem (vienu reizi darbojoties ar katru skaitli vai elementu).

Att. 7.1. Cikla *for* loģiskā struktūra

7.1.1. Cikls *for*, apstrādājot veselu skaitļu intervālu

Intervāla norādīšanai tiek izmantota funkcija *range*.

Nākošais *for* cikla piemērs (ar skaitļu intervālu) parāda skaitļu 1..*n* summas izrēķināšanu (ievērojiet, ka intervāls tiek uzstādīts no pirmā elementa ieskaitot, līdz pēdējam neieskaitot, tātad līdz a+1, nevis a):

```
a = int(input())
s = 0
for i in range(1,a+1):
    s += i
print(s)
```

```
9
45
```

Att. 7.2. Intervāla pārstaigāšana ar *for* ciklu

Funkcija *range* darbojas ar vienu, diviem vai 3 parametriem (Tab. 7.1).

Tab. 7.1.

Funkcijas *range* izmantošanas varianti

***range* variants**	**nozīme**
range (n)	visi skaitļi intervālā 0..n-1
range (a,b)	visi skaitļi intervālā a..b-1
range (a,b,s)	visi skaitļi intervālā a..b-1, sākot ar a, ik pa s
range (a,b,s), s<0, a>b	visi skaitļi intervālā a..b+1, sākot ar a, ik pa s atpakaļ

Ceturtā *range* varianta piemērs (ievērojiet – neieskaitot 5, skatoties no "augšas"):

```
for i in range(10,5,-2):
    print(i)
```

```
10
8
6
```

Att. 7.3. Intervāla pārstaigāšana ar *for* ciklu, izmantojot soli

7.1.2. Cikls *for*, apstrādājot elementu kopumu

Elementu kopums ir kāda datu struktūra – saraksts, vārdnīca, teksts (par struktūrām tiks aprakstīts vēlāk).

Cikls vārdu virknes specifiskai izdrukai:

```
letters = ["alpha","beta","gamma"]
for let in letters:
    print("Hello,"+let+"!")
```

```
Hello, alpha!
Hello, beta!
Hello, gamma!
```

Att. 7.4. Elementu kopuma pārstaigāšana ar *for* ciklu

Funkcija *enumerate* palīdz elementu kopuma pārstaigāšanā, automātiski uzturot elementa kārtas numuru:

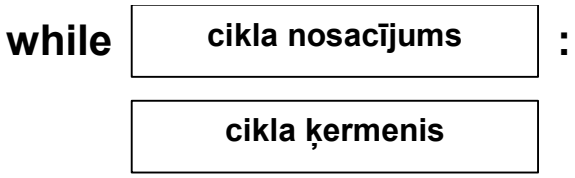
```
letters = ["alpha","beta","gamma"]
for num,let in enumerate(letters):
    print(num,"Hello, "+let+"!")
```

```
0 Hello, alpha!
1 Hello, beta!
2 Hello, gamma!
```

Att. 7.5. Elementu kopuma pārstaigāšana ar *for* ciklu, izmantojot *enumerate*

7.2. Cikls ar priekšnosacījumu while

Cikls *while* ir cikls ar nosacījumu, kas var būt gan saistīts ar skaitīšanu (tātad funkcionāli ietver to, ko dara for), gan kaut kas vispārīgāks un sarežģītāks.



Att. 7.6. Cikla *while* loģiskā struktūra

Nākošā programma ļauj no klaviatūras ievadīt veselus pozitīvus skaitļus un beigās izvada iepriekš ievadīto skaitļu summu. Kā ievadīšanas beigu pazīme kalpo nulles vai negatīva skaitļa ievadīšana:

```
s = 0
a = int(input())
while a>0:
    s += a
    a = int(input())
print(s)
```

```
6
2
7
0
15
```

Att. 7.7. *While* cikla piemērs skaitļu ievadīšanai no klaviatūras

7.3. Operatori break un continue un cikla else zars

Abiem nosauktajiem cikliem – *for* un *while* – cikla (turpināšanas) nosacījuma pārbaude notiek vienā punktā – cikla galvā (vai kā piederības intervālam/virknei pārbaude vai nosacījuma pārbaude). Šāda noteiktība nodrošina mazāku kļūdu ielaišanas iespējamību, veidojot ciklus, tomēr atsevišķos gadījumos, lai nodrošinātu īsāku un ērtāku pierakstu, būtu pieļaujams atkāpties no šī principa, izmantojot operatoru *break*, bez tam dažkārt ļoti ērts izmantošanai ir arī operators *continue*.

break

Operators *break* nodrošina kārtējās cikla iterācijas pārtraukumu un izeju no cikla.

continue

Operators *continue* nodrošina kārtējās cikla iterācijas pārtraukumu un pāreju uz nākošo iterāciju, izlaižot atlikušās cikla ķermeņa daļas izpildi.

Tātad *break* pārtrauc visu ciklu, bet *continue* tikai kārtējo tā iterāciju (pārlecot uz nākamo).

7.3.1. break

Operators *break* tiek izmantots izejai no cikla.

Nākošā programma parāda *break* kā alternatīvu cikla nosacījumam, kas nebūtu ieteicamais pielietojuma veids (Cikla nosacījums *True* nosaka t.s. mūžīgo ciklu).

```
# Skaitļu 1..5 summas izdrukāšana parastajā veidā
s = 0
i = 1
while i<=5:
    s += i
    i += 1
print(s)

# Skaitļu 1..5 summas izdrukāšana ar break cikla nosacījuma
                  vietā
s = 0
i = 1
while True:
    s += i
    if i==5: break
    i += 1
print(s)
```

```
15
15
```

Att. 7.8. *break* pilnībā aizvieto cikla nosacījumu – parasti tas nav labs stils

Tomēr tipiskākā situācija *break* lietošanai ir, ka nosacījums izejai no cikla tiek sadalīts starp cikla nosacījumu un *break* nosacījumu, piemēram, cikla nosacījumu lietojot skaitīšanai, bet *break* – ievades nosacījuma pārbaudei, tādējādi padarot katru nosacījumu vienkāršāku, nekā būtu viens liels kopējais nosacījums. Nākošais piemērs prasa no klaviatūras ievadīt līdz 5 skaitļiem un izvada to summu, tomēr, ja ievadā saņemts -1, tad summēšana beidzas ātrāk, nesasniedzot 5 skaitļus.

```
# 5 ievadīto skaitļu summas aprēķins ar pārtraukšanu pie -1
s = 0
i = 0
while i<5:
    n = int(input())
    if n==-1: break
    s += n
    i += 1
print(s)
```

```
1
```

```
2
3
4
5
15
```

```
1
2
3
-1
6
```

Att. 7.9. *break* nodrošina tikai daļu no nosacījuma iziešanai no cikla

Šādā uzdevumā savukārt *break* neizmantošana (Att. 7.10) padara cikla nosacījumu sarežģītāku, turklāt pārbaude uz ievades beigām (n!=1) vienalga jāliek arī cikla iekšienē.

```
# 5 ievadīto skaitļu summas aprēķins ar pārtraukšanu pie -1,
# neizmantojot break
s = 0
i = 0
n = 0
while i<5 and n!=-1:
    n = int(input())
    if n!=-1:
        s += n
        i += 1
print(s)
```

```
1
2
3
4
5
15
```

```
1
2
3
-1
6
```

Att. 7.10. Neizmantojot *break* situācijā, kur tas noderētu

7.3.2. for-else, while-else

Valodā *Python* arī cikliem (gan *for*, gan *while*) tāpat kā *if* ir pieejams *else* zars, kurš ciklu gadījumā izpildās, ja no cikla nav iziets ar *break* – tātad, tāds specializēts kods aiz cikla.

```
# Pāra skaitļu summas izvade no saraksta,
# ja visi ir pāra skaitļi, citādi nedrukā neko
s = 0
for a in [2,4,6,8]:
    s += a
    if a%2==1: break
else: print(s)
print("Pēc cikla")
```

```
20
Pēc cikla
```

Att. 7.11. *for-else*, kur *else* zars nostrādā

```
# Pāra skaitļu summas izvade no saraksta,
# ja visi ir pāra skaitļi, citādi nedrukā neko
s = 0
for a in [2,4,6,9]:
    s += a
    if a%2==1: break
else: print(s)
print("Pēc cikla")
```

```
Pēc cikla
```

Att. 7.12. *for-else*, kur *else* zars nenostrādā, jo no cikla iziet ar *break*

7.3.3. continue

Operators *continue* nodrošina kārtējās cikla iterācijas pārtraukumu un pāreju uz nākošo iterāciju, izlaižot atlikušās cikla ķermeņa daļas izpildi.

```
s = 0
for n in range(10,35):
    if n%10!=0: continue
    s += n
    print(n)
print(s)
```

```
10
20
30
60
```

Att. 7.13. *continue* pielietojums – izdukāt un saskaitīt skaitļus intervālā, kas dalās ar 10

Operatoru *continue* lieto daudz retāk nekā *break*, jo ir daudz mazāk tādu gadījumu, kad *continue* uzlabo programmas lasāmību – *continue* tikai "izslēdz no spēles" iterācijas beigu daļu, kas tāpat jādara ar *if* nosacījumu, vienīgi *continue* atļauj atlikušo iterācijas daļu nelikt vienu hierarhijas līmeni dziļāk.

```
s = 0
for n in range(10,35):
    if n%10==0:
        s += n
```

```
            print(n)
    print(s)
```

```
10
20
30
60
```

Att. 7.14. Izdrukāt un saskaitīt skaitļus intervālā, kas dalās ar 10 – bez *continue* (programmā ir par vienu hierarhijas līmeni vairāk)

8. Saraksti (*list*)

Saraksts valodā Python atbilst masīva jēdzienam citās programmēšanas valodās.

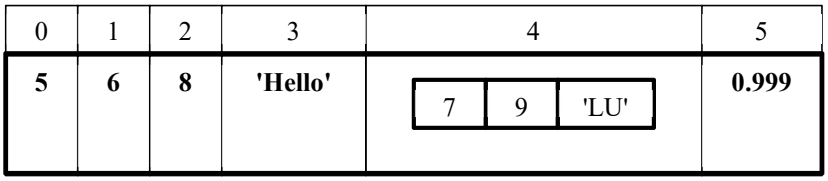
> **Saraksts (list)** ir indeksēta elementu virkne.

Vienu saraksta vienību sauc par **elementu**. Katru elementu identificē **indekss** – skaitlis, kas norāda, pie kura no elementiem vēršas lietotājs.

Tādējādi, vienu saraksta elementu identificē:

- saraksta vārds,
- indekss.

Saraksta indeksācija notiek pēc kārtas, bet valodā *Python* – vienmēr, sākot ar skaitli 0. Tādējādi, sarakstā ar garumu n tā elementi tiek indeksēti intervālā 0..n-1. Saraksts var sastāvēt no dažādu tipu elementiem vienlaikus, t.sk., citiem sarakstiem (sk. Att. 8.1, Att. 8.2). Saraksta uzdošanai, kā arī piekļuvei saraksta elementiem izmanto kvadrātiekavu notāciju [].



Att. 8.1. Saraksta piemērs shematiski (attiecīgo kodu sk. Att. 8.1)

```
aa = [5,6,8,'Hello',[7,9,'LU'],0.999]
print(aa)
print(aa[1])
print(aa[4])
print(aa[4][2])
aa[2] = 11
print(aa)
```

```
[5, 6, 8, 'Hello', [7, 9, 'LU'], 0.999]
6
[7, 9, 'LU']
LU
[5, 6, 11, 'Hello', [7, 9, 'LU'], 0.999]
```

Att. 8.2. *Python* saraksts un piekļūšana tā elementiem

8.1. Saraksta izveidošana

Tāpat kā ar parastajiem mainīgajiem – arī sarakstu nevis izveido un tad aizpilda, bet **izveido reizē ar aizpildīšanu**.

Izšķir šādus saraksta izveidošanas veidus:

1. tukša saraksta izveidošana ([], *list*()),
2. saraksta izveidošana ar inicializācijas virkni [...],
3. saraksta izveidošana pēc cita saraksta (vai citas datu struktūras) – pilna vai daļēja kopēšana ([:], *list*(·), *deepcopy*(·)),
4. saraksta izveidošana ar ģeneratoru.

Tukšu sarakstu izveido vai nu, izmantojot funkciju *list*, vai ar tukšām kvadrātiekavām [].

```
aa = []
bb = list()
print(aa, len(aa))
print(bb, len(bb))
```

```
[] 0
[] 0
```

Att. 8.3. Tukša saraksta izveidošana (funkcija *len* atgriež saraksta garumu)

Ar komatiem atdalītu vērtību virkne kvadrātiekavās automātiski nozīmē saraksta izveidošanu.

```
aa = [1,2,3,4,5]
print(aa)
```

```
[1, 2, 3, 4, 5]
```

Att. 8.4. Saraksta izveidošana ar inicializācijas virkni

Kopēšanu (informācijas dublēšanu) var veikt dažādos veidos – ar funkciju *list*, ar intervāla operatoru :, ar speciālo *deepcopy* funkciju, bet nekādā gadījumā, ne vienkārši piešķirot vienam mainīgajam otru (par piešķiršanu un kopēšanu (seklo un dziļo) vairāk aprakstīts sadaļā 9).

```
import copy
aa = [1,2,3,4,5]
bb = list(aa)
cc = aa[:]
dd = copy.deepcopy(aa)
ee = aa # Nav kopēšana! ee norāda uz to pašu sarakstu
aa[2] = 999 # mainām aa, lai saprastu, vai tas tika kopēts
print(bb)
print(cc)
print(dd)
print("Tas pats aa, nevis kopija:", ee)
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
Tas pats aa, nevis kopija: [1, 2, 999, 4, 5]
```

Att. 8.5. Saraksta pilnas kopijas izveidošana no cita saraksta

```
aa = list('Hello')
print(aa)
```

```
['H', 'e', 'l', 'l', 'o']
```

Att. 8.6. Saraksta izveidošana pēc simbolu virknes

Saraksta izveidošanai var lietot arī t.s. ģeneratorus. Saraksta ģenerators ir uz *for* cikla bāzēta konstrukcija, kas katrā iterācijā apraksta vienu vērtību, ko pievienot sarakstam – ja parastā *for* cikla gadījumā cikla ķermenis ir aiz *for* cikla galvas, tad ģeneratora gadījumā – ģenerējamā vērtība katrā solī ir aprakstīta pirms atslēgas vārda *for*. (Plašākā nozīmē ģenerators ir konstrukcija, kas katrā solī, pie tās vēršoties, atgriež nākošo vērtību. Par ģeneratoru funkcijām un ģeneratoru (jeb iterējošiem) objektiem sk. tālākajās sadaļās: 15.3, 16.6.3.)

```
aa = [(i+1)**2 for i in range (10)]
print(aa)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Att. 8.7. Saraksta izveidošana ar ģeneratoru – veselu skaitļu kvadrāti

8.2. Piekļūšana saraksta elementiem

Piekļūšana saraksta elementiem var notikt vai nu pa vienam elementam vai uzreiz noteiktam saraksta fragmentam, kas tādā gadījumā nozīmē saraksta vai tā fragmenta kopēšanu.

Izšķir šādus piekļūšanas veidus saraksta elementiem:

1. piekļūšana vienam elementam ar [i],
2. piekļūšana elementu intervālam a..b lasīšanas režīmā (iegūstot intervāla kopiju):
   a. tieši ar [a:b]
   b. specializēti ar soli s (reversi un/vai izlaižot elementus) – [a:b:s].
3. elementu intervāla modificēšana (tieši vai specializēti) – [a:b:s], bet piešķiršanas operatora kreisajā pusē.

Piekļūšana vienam elementam iespējama, ne tikai relatīvi pret sākumu, bet arī pret beigām (tādā gadījumā indeksu norādot negatīvu).

```
aa = [1,2,3,4,5,6,7,8]
print(aa[2])
print(aa[-2]) # otrais no beigām
aa[1]=999
print(aa)
```

```
3
7
[1, 999, 3, 4, 5, 6, 7, 8]
```

Att. 8.8. Piekļūšana vienam elementam lasīšanas un rakstīšanas režīmos

Python kontrolē, vai nav pārkāptas saraksta robežas.

```
aa = [1,2,3,4,5,6,7,8]
print(aa[10]) # indekss par lielu
```

```
Traceback (most recent call last):
  File "C:/src/list.py", line 2, in <module>
    print(aa[10])
builtins.IndexError: list index out of range
```

Att. 8.9. Python kļūdas paziņojums pie pārkāptas robežas

Saraksta intervālu iegūst, izmantojot intervāla operatoru :, tā tiek veidota intervāla kopija (saistībā ar intervāla operatoru sk. arī *range* funkciju nodaļā 7.1.1, kuras izmantošanas princips intervāla uzrādīšanai ir līdzīgs).

```
aa = [1,2,3,4,5,6,7,8]
print(aa[2:5]) # [a:b] - no a līdz b neieskaitot
print(aa[:5]) # ja a neuzrāda, tad no sākuma
print(aa[2:]) # ja b neuzrāda, tad līdz beigām
print(aa[:]) # visa saraksta izdrukāšana, pirms tam izveidojot
                  tā kopiju
```

```
[3, 4, 5]
[1, 2, 3, 4, 5]
[3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
```

Att. 8.10. Saraksta intervāla iegūšana

Intervālu var nodefinēt, neņemot visus elementus pēc kārtas vai pat pretējā virzienā, un to dara, norādot soli kā trešo intervāla operatora : parametru (tāpat kā funkcijai *range* nodaļā 7.1.1).

```
aa = [1,2,3,4,5,6,7,8]
print(aa[1::2]) # katru otro, sākot ar pozīciju 1
print(aa[::-1]) # apgriezti
print(aa[::-2]) # katru otro apgriezti
```

```
[2, 4, 6, 8]
[8, 7, 6, 5, 4, 3, 2, 1]
[8, 6, 4, 2]
```

Att. 8.11. Saraksta specializēta intervāla iegūšana (ar soli)

Intervāls piešķiršanas operatora kreisajā pusē nozīmē intervāla pārrakstīšanu

```
aa = [1,2,3,4,5]
aa[1:4] = [22,33,44]
print(aa)
```

```
[1, 22, 33, 44, 5]
```

Att. 8.12. Piekļuve intervālam rakstīšanas režīmā

Piešķiršana intervālam iespējama arī specializēti – ne pēc kārtas vai otrādi.

```
aa = [1,2,3,4,5,6,7]
aa[1:6:2] = [22,33,44]
print(aa)
```

```
[1, 22, 3, 33, 5, 44, 7]
```

Att. 8.13. Piekļuve intervālam rakstīšanas režīmā ar soli

8.3. Saraksta elementu pārlasīšana un piederības pārbaude

8.3.1. Saraksta elementu pārlasīšana

Saraksta elementu pārlasīšana var notikt šādos veidos (sk. arī nodaļu 7.1):

- tieša elementu iterācija – *for elem in elemlist...*,
- pārstaigāšana pēc indeksa – *for i in range(len(elemlist))...*,
- elementu iterācija ar *enumerate* – *for i,elem in enumerate(elemlist)...*

Tiešā iterācija ir visvienkāršākais elementu pārlasīšanas veids, to nodrošina *for* cikls.

```
aa = [1,2,3,4,5]
for a in aa:
    print(a)
```

```
1
2
3
4
5
```

Att. 8.14. Saraksta elementu pārlasīšana ar tiešo iterāciju

Diemžēl šādā veidā nevar nodrošināt datu elementu nomaiņu sarakstā, kas ir redzams nākošajā piemērā.

```
aa = [1,2,3,4,5]
for a in aa:
    a += 10
    print(a)
print(aa) # nemainīts
```

```
11
12
13
14
15
[1, 2, 3, 4, 5]
```

Att. 8.15. Tiešā iterācija nenodrošina iespēju nomainīt saraksta elementus

Alternatīva ir pārlasīt elementu indeksus, kas ļauj piekļūt elementiem arī nomainīšanas režīmā.

```
aa = [1,2,3,4,5]
for i in range(len(aa)):
    aa[i] += 10
print(aa)
```

```
[11, 12, 13, 14, 15]
```

Att. 8.16. Saraksta pārstaigāšana pēc indeksa

Funkcija *enumerate* palīdz elementu kopuma pārstaigāšanā, automātiski uzturot elementa kārtas numuru (tātad, nav jāizmanto *len*):

```
aa = [1,2,3,4,5]
sum = 0
for i,a in enumerate(aa):
    sum += a
    aa[i] += 10
print(aa, sum)
```

```
[11, 12, 13, 14, 15] 15
```

Att. 8.17. Saraksta pārstaigāšana, izmantojot *enumerate*

Kā redzams iepriekšējā piemērā, *for* cikls iterē ar diviem mainīgajiem *i* un *a*. Ir iespējams šos abus skaitļus uztvert kā vienotu struktūru – t.s. slēgto sarakstu garumā 2, ar elementu numuriem 0 un 1, kā redzams nākamajā piemērā. Par slēgto sarakstu jeb kortežu (*tuple*) vairāk nākamajā nodaļā.

```
aa = [1,2,3,4,5]
sum = 0
for item in enumerate(aa):
    sum += item[1]
    aa[item[0]] += 10
print(aa, sum)
```

```
[11, 12, 13, 14, 15] 15
```

Att. 8.18. Saraksta pārstaigāšana, izmantojot *enumerate* un slēgto sarakstu (*tuple*) viena elementa apzīmēšanai

8.3.2. Vērtību piederības pārbaude sarakstam (*in*, *not in*)

Saraksta elementa piederības pārbaude pēc vērtības notiek, izmantojot *in* un *not in* operatorus.

```
aa = [1,2,3,4,5]
print(2 in aa)
print(22 in aa)
print(2 not in aa)
print(22 not in aa)
```

```
True
False
False
True
```

Att. 8.19. *in* un *not in* elementa piederības pārbaudei sarakstā

Saraksta elementa piederību parasti izmanto lēmuma pieņemšanā (Att. 8.20).

```
aa = [1,2,3,4,5]
a = int(input())
if a in aa:
    print('IR')
else:
    print('nav')
if a not in aa:
    print('NAV')
else:
    print('ir')
```

```
2
IR
ir
```

```
99
nav
NAV
```

Att. 8.20. *in* un *not in* elementa piederības pārbaude ar lēmuma pieņemšanu

Ja piederības pārbaude jāveic bieži, tad saraksts (*list*) nav piemērotākā struktūra šim nolūkam, tā vietā labāk izmantot vārdnīcu (*dict*) vai kopu (*set*).

8.4. Elementu pievienošana vai izdzēšana no saraksta

8.4.1. Saraksta izmaiņa par vienu elementu

Saraksta izmēra izmaiņa par vienu elementu var notikt šādos veidos:

- *append(elem)* – elementa pievienošana beigās
- *insert(pos,elem)* – elementa pievienošana pēc pozīcijas
- *elem = pop()* – elementa izmešana no beigām (ar tā atgriešanu)
- *elem = pop(pos)* – elementa izmešana pēc pozīcijas (var būt negatīva)
- *remove(val)* – elementa (pirmā sastaptā) izmešana pēc vērtības

```
aa = [1,2,3,4,6,5,6]
aa.append('Added'); # pievieno beigās
print(aa)
out = aa.pop() # izmet no beigām
print(out,aa)
out = aa.pop(1) # izmet 1. no kreisās
print(out,aa)
aa.insert(3,'Hello') # ieievieto pirms 3. no kreisās
print(aa)
aa.remove(6) # Elementa izmešana pēc vērtibas
print(aa) # remove izmet tikai pirmo pēc dotās vērtības
```

```
[1, 2, 3, 4, 6, 5, 6, 'Added']
Added [1, 2, 3, 4, 6, 5, 6]
2 [1, 3, 4, 6, 5, 6]
[1, 3, 4, 'Hello', 6, 5, 6]
[1, 3, 4, 'Hello', 5, 6]
```

Att. 8.21. Saraksta izmaiņa par vienu elementu

8.4.2. Funkcija *del* viena vai vairāku saraksta elementu izdzēšanai

Funkcija *del* darbojas līdzīgi kā piešķiršana – var izdzēst vienu elementu, intervālu vai intervālu specializēti:

```
aa = [1,2,3,4,5,6]
del aa[2] # izdzēš vienu elementu
print(aa)
aa = [1,2,3,4,5,6]
del aa[2:5] # izdzēš intervālu
print(aa)
aa = [1,2,3,4,5,6]
del aa[2:5:2] # izdzēš intervālu specializēti (katru otro)
print(aa)
```

```
[1, 2, 4, 5, 6]
[1, 2, 6]
[1, 2, 4, 6]
```

Att. 8.22. Elementu izdzēšana ar *del*

8.5. Vairāku sarakstu apvienošana

8.5.1. Sarakstu konkatenācija

Sarakstu konkatenācija ir sarakstu pievienošana vienu otra galā, tādā veidā izveidojot lielāku sarakstu. Valodā *Python* tas iespējams šādos veidos:

- parastā konkatenācija (+),
- konkatenācijas ar piešķiršanu (konkatenācija sev) (+=)
- saraksta pavairošana (*)

Parastā konkatenācija (+) nodrošina viena vai vairāku sarakstu savienošanu, izmantojot operatoru.

```
bb = [5,6]
aa = [1,2,3] + bb + ['A','B'] # konkatenācija
print(aa)
```

```
[1, 2, 3, 5, 6, 'A', 'B']
```

Att. 8.23. Parastā konkatenācija

Konkatenācija ar piešķiršanu (+=) piekonkatenē sarakstu klāt esošam sarakstam, nevis veido jaunu trešo.

```
aa = [1,2,3]
bb = [5,6]
aa += bb # konkatenācija sev
print(aa)
```

```
[1, 2, 3, 5, 6]
```

Att. 8.24. Konkatenācija ar piešķiršanu

Pavairošana (*) veic vairākkārtēju viena un tā paša saraksta konkatenāciju.

```
print(['A','B','C'] * 3)
```

```
['A', 'B', 'C', 'A', 'B', 'C', 'A', 'B', 'C']
```

Att. 8.25. Sarakstu pavairošana

8.5.2. Vairāku sarakstu apvienošana pēc rāvējslēdzēja principa

Vairāku vienāda garuma sarakstu apvienošana pēc rāvējslēdzēja principa nozīmē viena saraksta iegūšanu ar to pašu garumu, kur katrā elementā ir visu oriģinālo sarakstu attiecīgās pozīcijas vērtības. To veic ar funkciju *zip* () (lai iegūtu sarakstu, vēl papildus jāpielieto funkcijā *list*). Katrs elements jauniegūtajā sarakstā ir slēgtais saraksts jeb kortežs (*tuple*) no attiecīgajiem elementiem (t.i. apaļajās, nevis kvadrātiekavās):

```
aa = [1,2,3]
bb = [11,22,33]
cc = [111,222,433]
dd = list(zip(aa,bb,cc))
print(dd)
```

```
[(1, 11, 111), (2, 22, 222), (3, 33, 433)]
```

Att. 8.26. Sarakstu apvienošana pēc rāvējslēdzēja principa ar *zip* – iegūts saraksts no sarakstiem

9. Piešķiršana, seklā kopēšana un dziļā kopēšana

Saraksta (vai citu objekta) piešķiršana var notikt šādos trīs veidos:

- parastā piešķiršana – cits mainīgais norāda uz jau eksistējošu sarakstu (jauns saraksts netiek veidots),
- seklā kopēšana (*shallow copying*) – tiek veidots jauns saraksts uz jau esoša saraksta bāzes (pilna vai daļēja kopija), bet tikai pirmajā līmenī,
- dziļā kopēšana (*deep copying*) – pilnas kopijas veidošana, dublējot visos līmeņos.

Ja saraksts satur vienkāršus elementus (piemēram, ja nesatur citus sarakstus kā elementus), tad dziļā kopēšana neatšķiras no seklās.

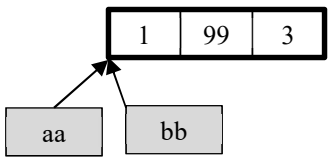
9.1.1. Saraksta parastā piešķiršana

Parastā piešķiršana neveido jaunu sarakstu, bet uzliek norādi uz to pašu sarakstu, un to var pārbaudīt ar operatoru *is*, kas pārbauda nevis vērtību, bet objektu identitāšu vienādību.

```
aa = [1,2,3]
bb = aa # bb norāda uz to pašu sarakstu
print(aa,bb)
print(aa is bb) # parāda, ka tas ir tas pats saraksts, nevis
                  vienkārši vienāds
aa[1] = 99 # mainām elementu
print(bb) # izmaiņas redzam arī caur mainīgo bb
```

```
[1, 2, 3] [1, 2, 3]
True
[1, 99, 3]
```

Att. 9.1. Parastā piešķiršana (sk. ilustrāciju Att. 9.2)



Att. 9.2. Saraksta piešķiršana (atbilstoši kodam Att. 9.1)

9.1.2. Seklā kopēšana

Seklā kopēšana (*shallow copying*) veido pilnu vai daļēju pirmā līmeņa kopēšanu. Ja sarakstā ir tikai viens līmenis, tad tas nozīmē kopēšanu pilnā dziļumā, respektīvi visas struktūras kopēšanu no līmeņu viedokļa.

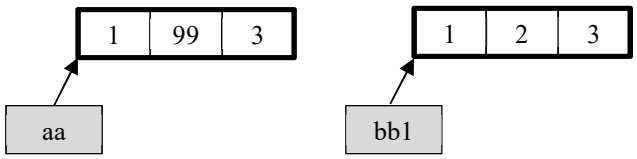
Seklo kopēšanu var veikt šādos veidos (Att. 9.3):

- ar funkciju *list*,
- ar intervāla operatoru (:),
- ar funkciju *copy*.

```
aa = [1,2,3]
bb1 = list(aa)
bb2 = aa[:]
bb3 = aa.copy()
bb4 = aa[:2]
print(aa is bb1, aa is bb2, aa is bb3, aa is bb4)
aa[1] = 999
print(aa,bb1,bb2,bb3,bb4)
```

```
False False False False
[1, 999, 3] [1, 2, 3] [1, 2, 3] [1, 2, 3] [1, 2]
```

Att. 9.3. Seklā kopēšana viena līmeņa sarakstā. Saraksti bb1..bb44 ir četras dažādas saraksta a sākotnējā varianta (pilnas vai daļējas) kopijas (sk. ilustrāciju Att. 9.4)



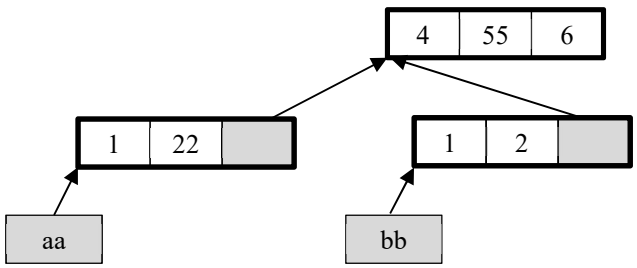
Att. 9.4. Saraksta seklā kopēšana (atbilstoši kodam Att. 9.3)

Seklā kopēšana kopē tikai pirmo līmeni, bet tālākie līmeņi netiek kopēti, bet tikai piešķirti

```
aa = [1,2,[4,5,6]]
bb = aa[:]
aa[1] = 22
aa[2][1] = 55
print(aa)
print(bb) # pirmā līmeņa kopija, bet otrajā tas pats masīvs
print(aa is bb) # nav tas pats saraksts
print(aa[2] is bb[2]) # otrajā līmenī ir tas pats saraksts
```

```
[1, 22, [4, 55, 6]]
[1, 2, [4, 55, 6]]
False
True
```

Att. 9.5. Seklā kopēšana vairāku līmeņu sarakstam. Tiek kopēts tikai pirmais līmenis (sk. Att. 9.6)



Att. 9.6. Saraksta seklā kopēšana vairāku līmeņu sarakstam (atbilstoši kodam Att. 9.5)

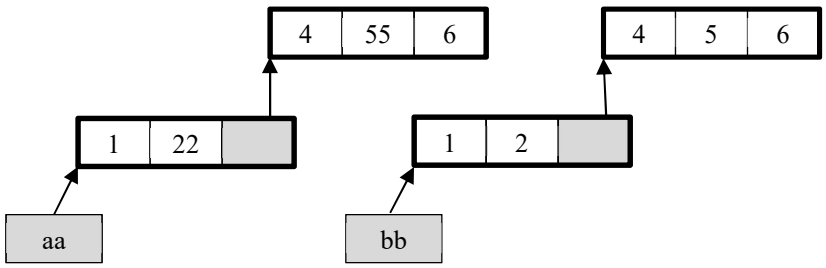
9.1.3. Dziļā kopēšana

Dziļā kopēšana (*deep copying*) kopē visus līmeņus, un to dara funkcija *deepcopy* no bibliotēkas *copy* (Att. 9.7).

```
import copy
aa = [1,2,[4,5,6]]
bb = copy.deepcopy(aa)
aa[1] = 22
aa[2][1] = 55
print(aa)
print(bb) # pilna kopija
print(aa is bb) # nav tas pats saraksts
print(aa[2] is bb[2]) # otrajā līmenī arī nav tas pats
                      saraksts
```

```
[1, 22, [4, 55, 6]]
[1, 2, [4, 5, 6]]
False
False
```

Att. 9.7. Dziļā kopēšana (sk. Att. 9.8)



Att. 9.8. Dziļā kopēšana (atbilstoši kodam Att. 9.7)

10. Slēgtie saraksti jeb korteži (*tuple*)

Valodā *Python* ir vēl viens saraksta veids – slēgtais saraksts (*tuple*), kas daudzējādā ziņā ir līdzīgs parastajam sarakstam (*list*), bet kam ir šādas atšķirības (vai nu pēc būtības, vai tehniski):

1. inicializācijas virkne tiek likta apaļajās iekavās, nevis kvadrātiekavās (Att. 10.1),
    a. lai inicializētu sarakstu garumā 1, aiz elementa liek papildus komatu ar nolūku atšķirt šo konstrukciju no parastas iekavu izmantošanas, piemēram, (99,), Att. 10.2.
2. datu tips saucas *tuple* (Att. 10.3),
3. slēgto sarakstu **nevar mainīt** (ne pievienojot vai izmetot elementus, ne mainot kāda elementa vērtību), Att. 10.4,
4. ja pēc līdzības ar sarakstu (*list*) slēgto sarakstu veido ar ģeneratoru, papildus jālieto funkcija *tuple*, citādi tiek izveidots t.s. ģeneratora objekts (Att. 10.5),
5. programmas tekstā par slēgto sarakstu dažreiz tiek interpretētas vērtības, kas vienkārši atdalītas ar komatiem (bez apaļajām iekavām apkārt), Att. 10.6.

Atšķirībā no saraksta (*list*), kas pieder t.s. maināmajiem (*mutable*) datu tipiem, slēgtais saraksts (*tuple*) pieder t.s. nemaināmajiem (*immutable*) – šīs ir divas svarīgas datu tipu kategorijas valodā *Python*.

Kāpēc vispār ir vajadzīgs datu tips *tuple*, ja ir datu tips *list*? Programmēšanā rekomendējamā prakse ir **"aizliegt visu, ko var aizliegt"**, ja vien tas netraucē, tādā veidā samazinot iespēju nejauši kaut ko mainīt un tādējādi pieļaut kļūdas (līdzīgiem nolūkiem citās programmēšanas valodās tiek lietoti modifikatori *const*, *private*), līdz ar to *tuple* ir tāds kā "drošais" saraksta variants, kurš noteikti būtu izmantojams, ja tas dotajā situācijā der.

```
t = (1,2,3) # tuple #1
u = (4,5,6) # tuple #2
print(t,u)
print(t+u) # slēgto sarakstu tuple konkatenācija
print(t[1:]) # tuple intervāla iegūšana
```

```
(1, 2, 3) (4, 5, 6)
(1, 2, 3, 4, 5, 6)
(2, 3)
```

Att. 10.1. Slēgtā saraksta *tuple* elementi inicializējot tiek liekti apaļajās iekavās, un ar slēgtajiem sarakstiem, ja vien tos nemaina, var veikt tās pašas darbības, ko ar parastajiem sarakstiem (*list*)

```
t = (1,) # tuple garumā 1
print(t)
print(t+(2,)) # pievieno otru garumā viens, izveidojot citu
                sarakstu, kas jau garumā 2
u = (4,5,6)
print(t[:1]) # paņem saraksta sākumu garumā 1
```

```
(1,)
(1, 2)
(1,)
```

Att. 10.2. Ja jāparāda slēgtais saraksts ar garumu 1, liek papildus komatu aiz vienīgā elementa

```
t = (1,2,3) # šis ir tuple
print(t, type(t))
u = [1,2,3] # šis ir list
print(u, type(u))
print(tuple(u), type(tuple(u))) # no list iegūstu tuple
```

```
(1, 2, 3) <class 'tuple'>
[1, 2, 3] <class 'list'>
(1, 2, 3) <class 'tuple'>
```

Att. 10.3. Slēgtā saraksta datu tips ir *tuple*

```
t = (1,2,3)
print(t[1])
t[1] = 22 # cenšos mainīt, bet neveiksmīgi
```

```
2Traceback (most recent call last):
  File "C:/src/datatypes.py", line 3, in <module>
    t[1] = 22
builtins.TypeError: 'tuple' object does not support item
                    assignment
```

Att. 10.4. Slēgto sarakstu nevar mainīt

```
t = (i**2 for i in range(10)) # tas nav tuple, bet ģenerators
print(t, type(t))
u = tuple(t) # tuple izveidošana, izmantojot ģeneratoru #1
print(u, type(u))
print(tuple((i**2 for i in range(10)))) # tuple izveidošana,
                    izmantojot ģeneratoru #2
```

```
<generator object <genexpr> at 0x03246ED0> <class 'generator'>
(0, 1, 4, 9, 16, 25, 36, 49, 64, 81) <class 'tuple'>
(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)
```

Att. 10.5. Slēgtā saraksta izveidošana ar ģeneratoru

```
t = 1,2 # arī bez iekavām tas ir tuple
print(t, type(t))
```

```
(1, 2) <class 'tuple'>
```

Att. 10.6. Par slēgto sarakstu (*tuple*) tiek interpretēti elementi, kas vienkārši atdalīti ar komatiem

11. Simbolu virknes

Simbolu virknes (*string*) ir simbolu secības, kas nodrošinātas ar dažādām funkcijām ērtākai teksta apstrādei.

Šī mācību materiāla ietvaros apskatīsim parasto (*Unicode*) simbolu virkni (datu tips: *str*), kas automātiski pieņem un glabā simbolus jebkurā kodējumā (valodā), un šīs visas ērtības nodrošināšanai *Python* var tērēt pat daudzus baitus uz katru simbolu (sk. Tab. 4.1). Tai pat laikā ir pieejami arī divi citi simbolu virkņu datu tipi – *bytes* un *bytearray* – kas ir ārpus šī materiāla sfēras.

Simbolu virkne (turpmāk ar to sapratīsim tieši datu tipu *str*) pieder pie nemaināmajiem datu tipiem (*immutable*), un to varētu tehniski salīdzināt ar slēgto sarakstu (*tuple*), sastāvošu tikai no simboliem, tomēr, ņemot vērā teksta kā informācijas formas lielo nozīmi, simbolu virknes specifika un papildus īpašības padara to īpašu citu datu tipu vidū, it sevišķi valodā *Python*, kur simbolu virkņu apstrādē ir ļoti plašas iespējas.

Simbolu virkni *Python* programmas tekstā parasti raksturo parastās vai dubultpēdiņas, kurās iekļauti simboli, bez tam simbolu virkni nevar mainīt, kā arī *Python* nav atsevišķi izdalīts tāds datu tips kā viens simbols.

```
s = 'Hello'
print(s,len(s))
s = "small brick ķieģelītis кирпичик μικρό τούβλο"
print(s,len(s))
```

```
Hello 5
small brick ķieģelītis кирпичик μικρό τούβλο 44
```

Att. 11.1. *Python* simbolu virkne liekama parastajās vai dubultpēdiņās, un tā strādā daudzās valodās

```
s = 'Hello'
print(s[1])
print(type(s[1]))
```

```
e
<class 'str'>
```

Att. 11.2. *Python* viens simbols nav atsevišķs datu tips, bet gan virkne garumā 1

```
s = 'Hello'
s[0]='h'
```

```
Traceback (most recent call last):
  File "C:/src/strings.py", line 2, in <module>
    s[0]='h'
builtins.TypeError: 'str' object does not support item
                    assignment
```

Att. 11.3. *Python* simbolu virkni nevar mainīt

```
s = 'Hello'
t = s # tā ir tā pati simbolu virkne, pieejama no cita
                  mainīgā
print(t is s) # vai abi mainīgie norāda uz vienu objektu - jā
s += ', World!' # tagad s jau ir cita virkne, nevis pamainīta
                  vecā
print(t is s) # vai abi mainīgie norāda uz vienu objektu - nē
print(s)
print(t)
```

```
True
False
Hello, World!
Hello
```

Att. 11.4. Šādi simbolu virknes maiņa it kā ir iespējama, bet patiesībā izveidojas jauns simbolu virknes objekts vecā objekta vietā

11.1. Simbolu virknes izveidošana

Tāpat kā ar parastajiem mainīgajiem – arī simbolu virkni nevis izveido un tad aizpilda, bet **izveido reizē ar aizpildīšanu**.

Izšķir šādus simbolu virknes izveidošanas veidus:

1. simbolu virknes izveidošana ar pēdiņām, parastajām vai dubultajām (' '," "),
2. simbolu virknes iegūšana, pārveidojot citu objektu ar funkciju *str*(·),
3. vairākrindu simbolu virknes izveidošana ar trīskāršajām pēdiņām, parastajām vai dubultajām (''' ''', """ """),
4. simbolu virknes izveide ar formatēšanu, izmantojot metodi *format* (sk. sadaļā 5.1.2. Izdrukājamās informācijas formatēšana).

```
s = 'Hello'
t = "Hello"
u = ''
print(s,t,len(u))
i = str(1234)
print(i+'5')
z = '''Hello,
World!'''
print(z)
```

```
Hello Hello 0
12345
Hello,
World!
```

Att. 11.5. Simbolu virknes izveidošana

11.2. Speciālie simboli

Vajadzība pēc speciālajiem simboliem var rasties šādos gadījumos:

- atdalītājsimbolu (piemēram, pēdiņām, kas rūpējas par pašas simbolu virknes atdalīšanu) norādīšanai,
- dienesta jeb neredzamo simbolu norādīšanai, piemēram, tabulācija, jaunas rindas simbols,
- simbolu norādīšanai, izmantojot tā kodu.

Speciālos simbolus norāda, izmantojot t.s. atsoļa simbolu (*escape character*) – atpakaļsvītru (*backslash*). Atpakaļsvītra simbolu sekvence nozīmē nevis atpakaļsvītru kā simbolu, bet gan iesāk speciālā simbola aprakstu, kas seko vienu (parasti) vai vairākas rakstzīmes aiz atpakaļsvītras (atkarībā no konteksta).

Ja simbolu virknē jāiekļauj viena veida atdalītājsimbols (tikai viena veida pēdiņas), tad var iztikt bez speciālajiem simboliem – atdalīšanai izmantojot otra veida pēdiņas.

```
s = '"Hello"'
print(s,len(s))
s = "'WORLD'"
print(s,len(s))
```

```
"Hello" 7
'WORLD' 7
```

Att. 11.6. Atdalītājsimbolu ievietošana simbolu virknē, neizmantojot speciālos simbolus

Tomēr, ja simbolu virknē vajag abus, tad bez speciālajiem simboliem (\' vai \") grūti iztikt.

```
s = '\"Hello\", \'WORLD\''
print(s,len(s))
```

```
"Hello", 'WORLD' 16
```

Att. 11.7. Atdalītājsimbolu norādīšana ar speciālajiem simboliem

Ja ar atdalītājsimboliem kaut kā var tikt galā (pat situācijā, ja vajag dažādus vienā simbolu virknē) bez speciālajiem simboliem, tad pašu apakšsvītru var norādīt tikai, izmantojot vēl vienu atpakaļsvītru kā atsoļa simbolu, tātad \\.

```
s = '"Hello", ' + "'WORLD'"
print(s,len(s))
s = "\\" # šis apzīmē tikai vienu simbolu
print(s,len(s))
s = "\š" # aiz atpakaļsvītras ir speciālajam simboliem
                nederīgs simbols, tāpec atpakaļsvītra "kļūst
                par" atpakaļsvītru viena pati
print(s,len(s))
```

```
"Hello", 'WORLD' 16
\ 1
\š 2
```

Att. 11.8. Atdalītājsimbolus var norādīt bez speciālajiem simboliem, bet pašai atpakaļsvītrai \ tas ir nepieciešams

No neredzamajiem simboliem tipiskākie ir jaunas rindiņas simbols \n un tabulācijas simbols \t (tabulācija kā simbols ir garāks tukšums).

```
s = "Hello,\nMy\tWorld!"
print(s,len(s))
```

```
Hello,
My    World! 16
```

Att. 11.9. Jaunas rindiņas simbols un tabulācijas simbols

Dažos gadījumos ir ērti pierakstīt simbolu virknes literāļus bez speciālas atsoļa simbola funkcionalitātes – ja simbolu virknē pašā ir atpakaļsvītra, piemēram, failu un direktoriju nosaukumos. Šādā gadījumā pirms simbola virknes literāļa liek burtu **r**, kas nozīmē *raw*.

```
f = r'c:\users\Documents\file.txt'
f
'c:\\users\\Documents\\file.txt'
print(f)
c:\users\Documents\file.txt
```

Att. 11.10. Jēlā (*raw*) simbolu virknes literāļa kodēšana – '\' netiek uztverts kā atsoļa simbols

Izmantojot speciālo simbolu *xNN*, simbols tiek norādīts ar heksadecimālu kodu.

```
s = "A<B"
print(s,len(s))
s = "\x41\x3c\x42"
print(s,len(s))
```

```
A>B 3
A<B 3
```

Att. 11.11. Simbolu virkne *A<B* parastā veidā un ar heksadecimālajiem kodiem

Ir arī citas iespējas norādīt simbola kodu, piemēram, izmantojot *Unicode* kodu.

11.3. Piekļūšana simbolu virknes elementiem

Piekļūšana simbolu virknes elementiem notiek tāpat kā sarakstam un to var darīt vai nu pa vienam elementam vai uzreiz noteiktam virknes fragmentam – vienīgi nav iespējams esošu virkni mainīt.

Izšķir šādus piekļūšanas veidus simbolu virknes elementiem:

1. piekļūšana vienam elementam ar [*i*],
2. piekļūšana elementu intervālam *a..b* lasīšanas režīmā (iegūstot intervāla kopiju):
    a. tieši ar [*a:b*]
    b. specializēti ar soli s (reversi un/vai izlaižot elementus) – [*a:b:s*].

Piekļūšana vienam elementam iespējama ne tikai relatīvi pret sākumu, bet arī pret beigām (tādā gadījumā indeksu norādot negatīvu).

```
s = 'DATORIKA'
print(s[2])
print(s[-2]) # otrais no beigām
```

```
T
K
```

Att. 11.12. Piekļūšana vienam elementam

Simbolu virknes saturu mainīt nedrīkst.

```
s = 'DATORIKA'
s[1] = 'O'
print(s)
```

```
Traceback (most recent call last):
  File "C:/src/strings.py", line 2, in <module>
    s[1]='O'
builtins.TypeError: 'str' object does not support item
                    assignment
```

Att. 11.13. Simbolu virkne *str* ir nemaināmais (*immutable*) datu tips

Python kontrolē vai nav pārkāptas virknes robežas.

```
s = [1,2,3,4,5,6,7,8]
print(s[10]) # indekss par lielu
```

```
Traceback (most recent call last):
  File "C:/src/strings.py", line 2, in <module>
    print(s[10]) # indekss par lielu
builtins.IndexError: list index out of range
```

Att. 11.14. *Python* kļūdas paziņojums pie pārkāptas robežas

Simbolu virknes intervālu iegūst, izmantojot intervāla operatoru :, tā tiek veidota intervāla kopija (saistībā ar intervāla operatoru sk. arī *range* funkciju nodaļā 7.1.1, kuras izmantošanas princips intervāla uzrādīšanai ir līdzīgs).

```
s = 'DATORIKA'
print(s[2:5]) # [a:b] - no a līdz b neieskaitot
print(s[:5]) # ja a neuzrāda, tad no sākuma
print(s[2:]) # ja b neuzrāda, tad līdz beigām
print(s[:]) # visas virknes izdrukāšana, pirms tam izveidojot
                    tā kopiju
```

```
TOR
DATOR
TORIKA
```

```
DATORIKA
```

Att. 11.15. Simbolu virknes intervāla iegūšana

Intervālu var nodefinēt, neņemot visus elementus pēc kārtas vai pat pretējā virzienā, un to dara, norādot soli kā trešo intervāla operatora : parametru (tāpat kā funkcijai *range* nodaļā 7.1.1).

```
s = 'DATORIKA'
print(s[1::2]) # katru otro, sākot ar pozīciju 1
print(s[::-1]) # apgriezti
print(s[::-2]) # katru otro apgriezti
```

```
AOIA
AKIROTAD
AIOA
```

Att. 11.16. Simbolu virknes specializēta intervāla iegūšana (ar soli)

11.4. Simbolu virknes elementu pārlasīšana pa vienam un piederības pārbaude

11.4.1. Virknes elementu pārlasīšana

Simbolu virknes elementu pārlasīšana pa vienam notiek tāpat kā sarakstam šādos veidos (sk. arī nodaļu 7.1):

- tieša elementu iterācija – *for elem in elemlist...*,
- pārstaigāšana pēc indeksa – *for i in range(len(elemlist))...*,
- elementu iterācija ar *enumerate* – *for i,elem in enumerate(elemlist)...*

Tiešā iterācija ir visvienkāršākais elementu pārlasīšanas veids, to nodrošina *for* cikls.

```
s = 'HELLO'
for a in s:
    print(a)
```

```
H
E
L
L
O
```

Att. 11.17. Simbolu virknes elementu pārlasīšana ar tiešo iterāciju

Alternatīva ir pārlasīt elementu indeksus.

```
s = 'HELLO'
for i in range(len(s)):
    print(s[i])
print(s)
```

```
H
E
L
```

```
L
O
HELLO
```

Att. 11.18. Simbolu virknes simbolu pārstaigāšana pēc indeksa

Funkcija *enumerate* palīdz elementu kopuma pārstaigāšanā, automātiski uzturot elementa kārtas numuru (tātad, nav jāizmanto *len*):

```
s = 'HELLO'
sum = 0
for i,a in enumerate(s):
    print(i,a)
```

```
0 H
1 E
2 L
3 L
4 O
```

Att. 11.19. Simbolu virknes pārstaigāšana, izmantojot *enumerate*

11.4.2. Vērtību piederības pārbaude simbolu virknei (*in*, *not in*)

Virknes elementa piederības pārbaude pēc vērtības notiek, izmantojot *in* un *not in* operatorus.

```
s = 'HELLO'
print('L' in s)
print('Z' in s)
print('L' not in s)
print('Z' not in s)
```

```
True
False
False
True
```

Att. 11.20. *in* un *not in* elementa piederības pārbaudei simbolu virknēā

11.5. Simbolu virkņu konkatenācija

Simbolu virkņu konkatenācija ir to pievienošana vienu otra galā, tādā veidā izveidojot lielāku simbolu virkni. Valodā *Python* tas iespējams šādos veidos:

- parastā konkatenācija (+),
- konkatenācija ar piešķiršanu (konkatenācija sev) (+=) (simbolu virkņu gadījumā, salīdzinot ar sarakstiem, gan izveidojas cita simbolu virkne),
- virkņu pavairošana (*)

Parastā konkatenācija (+) nodrošina vienas vai vairāku virkņu savienošanu, izmantojot operatoru +.

```
bb = ', '
aa = 'Hello' + bb + 'World!' # konkatenācija
print(aa)
```

```
Hello, World!
```

Att. 11.21. Parastā konkatenācija ar +

Parastā konkatenācija iespējama arī, vienkārši rakstot vairākas simbolu virknes blakus vienu otrai, vienīgi tām jābūt kā literāļiem, nevis izteiksmēm.

```
aa = 'Hello' ', ' 'World!' # konkatenācija netieši
print(aa)
```

```
Hello, World!
```

Att. 11.22. Netiešā konkatenācija bez +, kas strādā tikai simbolu virknē tiešā tekstā

Konkatenācija ar piešķiršanu (+=) piekonkatenē sarakstu "klāt" esošai simbolu virknei (tā izskatās, un algoritmiskā nozīmē lielā daļā gadījumu tā "drīkst" domāt), kas patiesībā nozīmē jauna objekta izveidošanu, jo *str* ir nemaināmais datu tips (*immutable*).

```
s = 'Hello'
t = s # tā ir tā pati simbolu virkne, pieejama no cita mainīgā
print(t is s) # vai abi mainīgie norāda uz vienu objektu - jā
s += ', World!' # tagad s jau ir cita virkne, nevis pamainīta
                  vecā
print(t is s) # vai abi mainīgie norāda uz vienu objektu - nē
print(s)
print(t)
```

```
True
False
Hello, World!
Hello
```

Att. 11.23. Konkatenācija ar piešķiršanu

Pavairošana (*) veic vairākkārtēju vienas un tās pašas virknes konkatenāciju.

```
print('ABC' * 3)
```

```
ABCABCABC
```

Att. 11.24. Simbolu virkņu pavairošana

11.6. Dažas specializētas metodes simbolu virkņu apstrādei

Bez jau iepriekš aprakstītajām metodēm *str*, *len* un *format* šeit tiks parādītas tikai dažas no ļoti daudzām metodēm, ko piedāvā *Python*:

- simbolu virknes apgriešana ar *strip*(), *lstrip*(), *rstrip*(),
- virknes saskaldīšana sarakstā ar *split*(),
- saraksta salīmēšana par virkni ar *join*(),
- virknes burtu reģistra maiņa ar *upper*(), *lower*(), *capitalize*(),

- simbola koda iegūšana, un koda iegūšana pēc simbola ar *chr*() un *ord*(),
- simbola veida pārbaude ar *isalpha*(), *isdigit*(), *isupper*(), *islower*()
- apakšvirknes meklēšana ar *find*(), *startswith*(), *endswith*(),
- apakšvirkņu nomaiņa ar *replace*(),

Cita starpā šī mācību materiāla ietvaros nav aprakstīts tāds spēcīgs mehānisms sarežģītai meklēšanai un izmaiņu izdarīšanai simbolu virknēs kā regulāras izteiksmes.

Simbolu virkni var apgriezt, novācot tukšuma simbolus (tukšums, tabulācija, jauna rindiņa utml.) no abām pusēm (*strip*) vai tikai no vienas (*lstrip*, *rstrip*).

```
s = '  a b  c   '
print('[' + s + ']') # oriģinālā virkne ar tukšumiem abās
                     pusēs
print('[' + s.strip() + ']') # apgriešana no abām pusēm
print('[' + s.lstrip() + ']') # no kreisās
print('[' + s.rstrip() + ']') # no labās
```

```
[  a b  c   ]
[a b  c]
[a b  c   ]
[  a b  c]
```

Att. 11.25. *strip*, *lstrip* un *lstrip* simbolu virknes apgriešanai (tukšo simbolu ērtākai pamanīšanai papildus tiek izdrukātas kvadrātiekavas)

Simbolu virkni saskalda apakšvirknēs ar *split* (apakšvirkņu robežas nosaka tukšuma simboli), un izveidojas saraksts no apakšvirknēm. Pretēja darbība notiek ar *join*, kas izveido simbolu virkni no saraksta ar virknēm.

```
s = '  a b  c   '
ss = s.split() # saskalda pēc tukšumiem
print(ss)
t = " : ".join(ss) # salīmejot uzrāda atdalītāju
print(t)
```

```
['a', 'b', 'c']
a : b : c
```

Att. 11.26. *split* simbolu virknes saskaldīšanai, un *join* - salīmēšanai

Reģistra maiņas trīs galvenās, bet ne vienīgās metodes ir nomainīt lielo burtu uz mazo (*lower*), mazo uz lielo (*upper*), virknes pirmo uz lielo, bet pārējos uz maziem (*capitalize*).

```
s = 'woRLd'
print(s)
print(s.lower())
print(s.upper())
print(s.capitalize())
```

```
woRLd
world
WORLD
World
```

Att. 11.27. Simbolu virknes burtu reģistra maiņa ar *lower*, *upper*, *capitalize*

Funkcija *ord* atgriež simbola kodu, bet *chr* tieši pretēji – simbolu pēc tā koda.

```
print(ord("A")) # burta kods
print(chr(66)) # burts pēc koda
```

```
65
B
```

Att. 11.28. *ord* un *chr* simbolu kodu aprēķinos

Metode *islower* nosaka, vai visi burti ir mazie, *isupper* – vai visi burti ir lielie, *isalpha* – vai visi ir burti, *isdigit* – vai visi ir cipari.

```
print("abc34d".islower(), "aBC34d".islower()) # visi burti
                  mazie
print("ABC34D".isupper(), "aBC34d".isupper()) # visi burti
                  lielie
print("ABCD".isalpha(), "aBC34d".isalpha()) # visi ir burti
print("34".isdigit(), "aBC34d".isdigit()) # visi ir cipari
```

```
True False
True False
True False
True False
```

Att. 11.29. Simbola veida pārbaude ar *islower*, *isupper*, *isalpha*, *isdigit*

Apakšvirknes meklēšana veicama ar *find* (atrod pozīciju, citādi -1), *startswith* (vai ir sākumā), *endswith* (vai ir beigās).

```
s = "hello my big world"
print(s.find('hello'),s.find('big')) # atrod pozīciju
print(s.find('WORLD')) # neatrod šadu apakšvirkni
print(s.startswith('hello'),s.startswith('other')) # vai sākas
print(s.endswith('world'),s.endswith('other')) # vai beidzas
```

```
0 9
-1
True False
True False
```

Att. 11.30. Apakšvirknes meklēšana ar *find*, *startswith*, *endwith*

Apakšvirknes nomainīšanu par citu veic ar *replace*.

```
s = "hello my big world"
t = s.replace(' ',' - ')
print(t)
```

```
0 9
-1
hello - my - big - world
```

Att. 11.31. Apakšvirknes nomainīšana ar *replace* (1. parametrs – meklējamā apakšvirkne, 2. parametrs – jaunā apakšvirkne)

12. Funkcijas

12.1. Funkcija kā programmas strukturēšanas līdzeklis

Viens no svarīgiem strukturētās programmēšanas pamatprincipiem ir t.s. procedurālā abstrakcija.

> **Procedurālā abstrakcija** ir programmas strukturēšanas veids, kad programma tiek strukturēta moduļos (procedūrās) pēc "melnās kastes" (*black box*) principa, t.i., nodalot redzamo vispārīgo (deklaratīvo) daļu no paslēptās detalizētās (realizācijas) daļas.

Ideālā gadījumā katrs modulis tiek noformēts tā, ka, lai ar to strādātu, nepieciešams zināt tikai to, kādi argumenti tam jāpadod un ka tas atgriezīs vajadzīgo rezultātu, bet nav nepieciešams zināt, kā tas uzbūvēts.

Valodā *Python* procedurālo abstrakciju nodrošina funkcijas.

Funkciju veidošana pēc "melnās kastes" principa bieži tiek saukta par **informācijas slēpšanu** (*information hiding*). Informācijas slēpšana ir svarīgs mehānismu kopums jebkurā augsta līmeņa programmēšanas valodā, un tai ir ļoti svarīga loma arī objektorientētajā programmēšanā, kas tiks apskatīta vēlāk.

> **Funkcija** ir patstāvīgs nosaukts programmas bloks, kuru iespējams izsaukt no citas koda daļas un kas ar padotajiem datiem veic noteiktas darbības un iespējami atgriež vērtību. Ārēji funkciju raksturo vārds un parametri.

Funkcijai

- var būt neviens vai vairāki parametri,
- funkcija var kaut ko atgriezt vai neatgriezt neko (*Python* gadījumā tas nozīmē – atgriež tukšo vērtību *None*).

Ar funkciju darbību cieši saistīti jēdzieni ir parametri un argumenti.

> **Parametri** ir mehānisms, ar kuru vērtības tiek nodotas funkcijai noteiktu darbību veikšanai.

> **Argumenti** ir vērtības, kas, funkciju izsaucot, tai tiek padotas caur parametriem.

Alternatīva terminoloģija. Gan parametrus, gan argumentus reizēm mēdz saukt vienkārši par parametriem, bet reizēm, lai tos atšķirtu, parametrus sauc par **formālajiem parametriem**, bet argumentus par **faktiskajiem** jeb **aktuālajiem parametriem**.

Valodā *Python* funkcijas aprakstīšanu (definēšanu, realizāciju) sāk ar atslēgas vārdu *def*, kam seko funkcijas vārds, parametru virkne un funkcijas ķermenis.

Līdzīgi kā ar mainīgajiem – funkcijai nav atgriežamā datu tipa un datu tipu nav funkcijas parametriem.

Nākošajā piemērā ir parādīta funkcija ar diviem parametriem a un b, kas rēķina šo divu skaitļu summu, bet, šo funkciju izsaucot galvenajā programmā, caur tiem tiek padoti argumenti x un y:

```
def add(a,b):
    return a+b

x=5
y=7
z=add(x,y)
print(z)
```

```
12
```

Att. 12.1. Programmas piemērs ar funkciju *add* divu veselu skaitļu saskaitīšanai

Kā redzams piemērā, funkcija sastāv no galvas un ķermeņa pakārtotā bloka veidā, kurā uzskaitītas darbības, kuras veic funkcija. Funkcijas galvā redzams atgriežamais atslēgas vārds def, funkcijas vārds (*add*) un funkcijas parametru nosaukumi (*a,b*).

Parādītajā piemērā mēs redzam funkcijas *add* realizāciju un pēc tam funkcijas *add* izsaukumu programmā.

Funkcijas ķermenī var tikt veidoti mainīgie, un tie būs t.s. lokālie mainīgie – izmantojami tikai funkcijas iekšienē. No funkcijas ķermeņa skatupunkta arī funkcijas parametri ir lokālie mainīgie. Atšķirībā, piemēram, no *C++*, valodā *Python* mainīgo lokālums ir visas funkcijas līmenī (nevis mazāka struktūras bloka līmenī).

12.2. Funkciju definēšana un izsaukšana

12.2.1. Galvenie principi

Funkcijas programmās tiek lietotas, lai strukturētu, kā arī koplietotu kodu.

Funkcijas parādīšanās programmā notiek divos veidos:

- funkcijas definīcija (jeb realizācija, implementācija) – funkcijas apraksts,
- funkcijas izsaukumi – komandas, kas liek darbināt doto funkciju.

Funkcijas definīcija nosaka šādas lietas par funkciju:

- funkcijas vārds,
- parametru kopums, kas nosaka, kā funkcija saņem datus,
- funkcijas ķermenis, kas netieši nosaka funkcijas atgriežamās vērtības (*Python* gadījumā nav tāda atgriežamā datu tipa)

Bez parametriem un atgriežamajām vērtībām funkcija ar pārējo programmu var apmainīties ar informāciju arī, izmantojot globālos mainīgos, t.i., tādus, kas izveidoti ārpus funkcijām.

Funkcijas izsaukumā tiek izmantoti visi vai daļa no parametriem, kas noteikti funkcijas definīcijā (daļa parametru ir uzstādāmi kā noklusētie).

Funkcija var būt bez parametriem un arī neko neatgriezt. *Python* funkcija vienmēr kaut ko atgriež – ja nekas nav noteikts, tad speciālo vērtību *None*.

```
def empty(): # funkcijas definīcijas sākums
    pass # tukšais operators

empty() # pirmais izsaukums
print(empty()) # otrais izsaukums, kas izdrukā None
```

```
None
```

Att. 12.2. Tukša funkcija ar diviem izsaukumiem

12.2.2. Funkciju parametri

Parametru izmantošanas veidi izsaucot. Argumentus (vērtības) caur parametriem, izsaucot funkciju, var padot divos galvenajos veidos (kas gan var arī tikt kombinēti):

- pozicionāli (pēc kārtas),
- pēc nosaukuma (t.sk. noklusētie jeb neobligātie).

Noklusētie (neobligātie) parametri. Ja netiek padotas vērtības caur noklusētajiem parametriem (tiem, kam funkcijas definīcijā ir norādītas vērtības), tad šie parametri pieņem šīs vērtības automātiski.

```
def divide(a,b=1):
    return a / b

print(divide(13,5)) # visu parametru norādīšana pozicionāli
                    (13/5)
print(divide(13)) # otrs parametrs pieņem noklusēto vērtību
                    (13/1)
print(divide(b=7,a=20)) # parametru norādīšana pēc nosaukuma
                    (20/7)
```

```
2.6
13.0
2.857142857142857
```

Att. 12.3. Vērtību padošana caur parametriem pozicionāli vai pēc nosaukuma, noklusētie parametri

Vērtības caur parametriem var padot **gan pozicionāli, gan pēc nosaukuma** – tādā gadījumā vispirms (no kreisās) tās jāpadod pozicionāli un tikai tad (pārējie) pēc nosaukuma.

```
def process(a,b,c):
    return (a+b)*c

y = process(7,c=2,b=3) # y = (7+3)*2
print(y)
```

```
20
```

Att. 12.4. Parametru izmantošana gan pozicionāli, gan pēc nosaukuma vienlaikus

Ja, funkciju izsaucot, parametru skaits (vai izsaukšanas secība) neatbilst definīcijai, tiek parādīta kļūda.

```
def add(a,b):
    return a+b

print(add(1,2,3))
```

```
Traceback (most recent call last):
  File "C:/src/add.py", line 4, in <module>
    print(add(1,2,3))
builtins.TypeError: add() takes 2 positional arguments but 3
                    were given
```

Att. 12.5. Parametru izmantošana gan pozicionāli, gan pēc nosaukuma vienlaikus

12.2.3. Funkcijas vērtības atgriešana

Funkcija var atgriezt vienu vai vairākas vērtības – bet ja neatgriež neko, tad atgriež speciālo vērtību *None*. Atgriežot vairākas vērtības, tās *return* operatora izsaukumā jāatdala ar komatiem

```
def intdivide(a,b):
    return a // b, a % b # gan dalījums, gan atlikums

d,r = intdivide(13,5)
print(d,r)
```

```
2 3
```

Att. 12.6. Funkcija, kas atgriež vairākas vērtības

Atgriežot vairākas vērtības, funkcija patiesībā tehniski atgriež vienu – slēgto sarakstu (*tuple*) no šīm vērtībām.

```
def intdivide(a,b):
    return a // b, a % b # gan dalījums, gan atlikums

dr = intdivide(13,5)
print(dr,type(dr))
```

```
(2, 3) <class 'tuple'>
```

Att. 12.7. Vairākas vērtības, kas atgrieztas kā *tuple*

Viena un tā pati funkcija var atkarībā no situācijas atgriezt dažādu skaitu vērtību.

```
def intdivide(a,b):
    if b==0: return "Kļūda"
    else: return a // b, a % b

print(intdivide(13,5))
print(intdivide(13,0))
```

```
(2, 3)
Kļūda
```

Att. 12.8. Dažādu tipu un skaita vērtību atgriešana atkarībā no izsaukuma

12.3. Funkciju papildus īpašības un iespējas

12.3.1. Funkcijas un globālie mainīgie

Globālie mainīgie ir tādi mainīgie, kas izveidoti ārpus funkcijas un tāpēc ir pieejami jebkurai funkcijai. Globālie mainīgie ir izmantojami informācijas apmaiņai starp funkcijām, bet vispārīgā gadījumā tas nav labs stils.

Globālie mainīgie funkcijā izmantojami divos režīmos:

- lasīšanas un modificēšanas,
- pārrakstīšanas režīmā

Lasīšanas/modificēšanas režīmā globālais mainīgais ir vienkārši izmantojams funkcijā, it kā tas būtu lokālais mainīgais. Pie šī režīma pieder arī maināma (*mutable*) datu tipa mainīgā modificēšana.

```
a = 5 # globālais mainīgais

def g1():
    print(a) # a pieejams kā lokālais mainīgais

g1()

aa = [1,2,3] # globālais mainīgais

def g2():
    aa[1]=222 # aa pieejams kā lokālais mainīgais
    print(aa)

g2()
print(aa) # aa ir ticis izmanīts funkcijā
```

```
5
[1, 222, 3]
[1, 222, 3]
```

Att. 12.9. Globālie mainīgie (*a*, *aa*) funkcijā lasīšanas/modificēšanas režīmā

Kolīdz "globālajam" mainīgajam funkcijā notiek piešķiršana, tā automātiski tiek izveidots lokālais mainīgais ar tādu pašu nosaukumu, bet globālais mainīgais netiek izmantots.

```
a = 5 # globālais mainīgais

def g3():
    a = 555 # tiek izveidots lokālais mainīgais a
    print(a) # izdrukā lokālo a

g3()
print(a) # globālais a nav mainījies
```

```
555
5
```

Att. 12.10. Lokālā mainīgā izveidošana ar piešķiršanu un globālā mainīgā ignorēšana

Ja "globālo" (nemaināma tipa) mainīgo funkcijā mēģina mainīt ar piešķiršanu (+=, *= utt.), tiek izmesta kļūda. Nākošajā piemērā *a+=1* nozīmē *a = a + 1*, tātad, piešķirot a vērtību, tiek automātiski veidots lokālais mainīgais *a*, nevis ņemts globālais, bet viņa aizpildīšanai (*a+1*) vajadzīgs jau eksistējošs lokālais mainīgais a.

```
a = 5 # globālais mainīgais

def g3():
    a += 1  # neļauj, jo pieprasa, ka lokālais a eksistē
    print(a)

g3()
print(a)
```

```
Traceback (most recent call last):
  File "C:/src/global.py", line 7, in <module>
    g3()
  File "C:/src/global.py", line 4, in g3
    a += 1  # neļauj, jo pieprasa, ka lokālais a eksistē
UnboundLocalError: local variable 'a' referenced before
                   assignment
```

Att. 12.11. Globālā mainīgā neveiksmīga mainīšana funkcijā

Lai globālo mainīgo izmantotu funkcijā pārrakstīšanas režīmā, tas iepriekš funkcijas iekšienē jādeklarē kā globālais, izmantojot atslēgas vārdu *global*.

```
a = 5 # globālais mainīgais

def g3():
    global a # deklarējam globālo mainīgo
    a = 555 # šis ir globālais
    print(a) # izdrukā globālo a

g3()
print(a) # globālais a tagad ir mainījies
```

```
555
555
```

Att. 12.12. Globālā mainīgā pārrakstīšana funkcijā

12.3.2. Funkcijas un references parametri

Lielā daļā programmēšanas valodu ir pieejami divu veidu parametri – vērtību parametri (*by value*), kur vērtība tiek dublēta funkcijas vajadzībām un references parametri (*by reference*), kur funkcijai tiek padota reference (norāde) uz padoto vērtību, un funkcijā tiek izmantota tā pati vērtība, nevis tās dublikāts.

Valodā *Python* parametri šādā nozīmē ir viena veida – daļēji vērtības parametri, daļēji references parametri, un to uzvedību nosaka tas, vai funkcijā parametra mainīgajam tiek piešķirta vērtība, vai nē (pēc tāda paša principa kā globālajiem mainīgajiem sadaļā 12.3.1):

- ja parametru funkcijā izmanto lasīšanas režīmā, vai caur to tiek modificēta maināma (*mutable*) vērtība, piemēram, *list*, tad tas strādā references režīmā (Att. 12.13),
- ja parametra mainīgajam tiek veikta piešķiršana, tad tiek veidota jauna vērtība, un parametrs "pārtop" par vērtības parametru (Att. 12.14).

```
a = 5

def g1(x):
    print(x)
    print(x is a) # tas ir tas pats globālais

g1(a)

aa = [1,2,3]

def g2(xx):
    xx[1]=222
    print(xx)
    print(xx is aa) # tas ir tas pats globālais

g2(aa)
print(aa) # aa ir ticis izmainīts funkcijā
```

```
5
True
[1, 222, 3]
True
[1, 222, 3]
```

Att. 12.13. Parametri (*x*, *xx*) funkcijā lasīšanas/modificēšanas režīmā kā references parametri (salīdzināt ar globālo mainīgo izmantošanu Att. 12.9)

```
a = 5

def g4(x):
    x = 555 # kļūst par citu vērtību
    print(x)
    print(x is a) # tas vairs nav tas pats globālais

g4(a)
print(a) # globālais a nav mainījies
```

```
555
False
5
```

Att. 12.14. Parametrs pārrakstīšanas režīmā kļūst par vērtības parametru (salīdzināt ar globālo mainīgo izmantošanu Att. 12.12)

Līdz ar to var teikt, ka maināmie (*mutable*) datu tipi, piemēram, *list*, *dict*, tiek nodoti kā references, bet nemaināmie, piemēram, *int*, *str*, kā references tiek nodoti tikai lasīšanas režīmā.

Ko darīt, ja gribas "izmainīt" nemaināma (*immutable*) datu tipa vērtību, padodot to caur parametru? Ir divi varianti, kas to risina, bet, protams, tieši nenozīmē izmainīšanu caur parametru:

- atgriezt ar *return* (*Python* valodā funkcija var atgriezt vairākas vērtības), Att. 12.15,
- ievietot maināmā datu tipā (piemēram, *list*) un tad padot to (šis gan ir tāds "samākslots" variants, ja izmainīšana caur parametru ir pašmērķis), Att. 12.16.

```
def inc(a,b):
    return a+1,b+1 # abas vērtības palielina par 1

a,b = 5,7
a,b = inc(a,b) # jaunās vērtības pārraksta pa virsu
print(a,b)
```

```
6 8
```

Att. 12.15. Nemaināma datu tipa izmainīšanas caur parametru modelēšana – *return* izmantošana

```
def inc(xx):
    xx[0]+=1

a = 5
aa = [a] # ievieto sarakstā
inc(aa) # maina sarakstu
a = aa[0] # atkal pārraksta pa virsu
print(a)
```

```
6
```

Att. 12.16. Nemaināma datu tipa izmainīšanas caur parametru modelēšana – ievietošana maināmā datu tipā

12.3.3. Funkciju definīciju un izsaukumu izvietojums programmā

Funkciju izvietošanai programmā ir šādi nosacījumi:

- funkcijas definīcijai jābūt pirms tās izsaukuma (Att. 12.17, Att. 12.18),
  - t.sk., drīkst veidot rekursīvas funkcijas, t.i., tādas, kas izsauc pašas sevi (Att. 12.19),
- drīkst ievietot funkciju (funkcijas definīciju) citā funkcijā (Att. 12.21),
- programmā (modulī) drīkst būt tikai viena funkcija ar šādu nosaukumu (valodā *Python* nav funkciju pārslogošanas (*overloading*)) (Att. 12.22, Att. 12.23).

Funkcijas definīcijai ir jābūt pirms izsaukuma.

```
print(fun())

def fun():
    return 999
```

```
Traceback (most recent call last):
  File "C:/src/fun.py", line 1, in <module>
    print(fun())
builtins.NameError: name 'fun' is not defined
```

Att. 12.17. Funkcijas definīcija nav pirms izsaukuma, tāpēc iestājas kļūda

Funkcijas definīcijai ir jābūt pirms funkcijas izsaukuma no izpildes viedokļa, nevis no programmas koda viedokļa.

```
def fun2():
    print(fun())

def fun():
    return 999

fun2()
```

```
999
```

Att. 12.18. Funkcijas definīcija (*fun*) nav pirms izsaukuma tekstā, bet no izsaukuma viedokļa ir (izsaukums sākas no funkcijas fun2 izsaukuma pēdējā rindiņā, kad abas funkcijas jau ir nodefinētas), tāpēc viss ir korekti

Rekursīvas funkcijas ir tādas funkcijas, kuras izsauc pašas sevi. Tās parasti lieto rekursīvu (vairāklīmeņu dinamisku) datu struktūru apstrādei, bet dažreiz (kā Att. 12.19, un tā parasti nav laba pieeja) arī kā alternatīvu ciklam.

Kā alternatīva ciklam rekursija ir slikta pieeja, jo katrs funkcijas izsaukums patērē atmiņu, tāpēc šāda pieeja (izmantot rekursiju cikla vietā) ir pieļaujama, ja izpildās šādas abas īpašības:

- iterāciju skaits nav liels (piemērā Att. 12.19 piemērā faktoriāls aug ātri, un soļu skaits nevar būt liels),
- rekursīvā pieeja uzlabo programmas lasāmību (piemērs Att. 12.20 parāda, ka šis "uzlabojums" faktoriāla gadījumā nav acīmredzams).

Rekursīvas funkcijas ķermenī ir divas daļas:

- bāzes daļa – kas nodrošina rekursijas apstāšanos,
- rekursijas daļa – kas turpina programmas izpildi, izsaucot to pašu funkciju (potenciāli – ar datiem, kas ir vienu soli tuvāk risinājumam).

```
def fact(n):
    if n<=1: return 1
    else: return n * fact(n-1)

print(fact(5))
```

```
120
```

Att. 12.19. Rekursīva funkcija faktoriāla izrēķināšanai

```
def fact(n):
    f = 1
    for i in range(2,n+1):
        f *= i
    return f

print(fact(5))
```

```
120
```

Att. 12.20. Faktoriāla izrēķināšana bez rekursijas

Funkcijas var ievietot arī vienu otrā – tādā gadījumā tās iegūst savu lokālo mainīgo redzamības apgabalu, kā arī piekļuvi mātes funkcijas mainīgajiem.

```
def digitsum2(a,b):
    def digitsum(x):
        sum = 0
        while(x>0):
            sum += x%10
            x //= 10
        return sum
    return digitsum(a)+digitsum(b)
print(digitsum2(123,456))
```

```
21
```

Att. 12.21. Funkcija ar iekļutu citu funkciju divu skaitļu ciparu summas aprēķināšanai

Valodā *Python* lietotāja veidotajām funkcijām nav iespējama pārslogošana, un tas nemaz nav vajadzīgs, pateicoties dinamiskajai tipu sistēmai.

```
def a():
    print("pirmā")

a()

def a():
    print("otrā")

a()
```

```
pirmā
otrā
```

Att. 12.22. Funkcijas ar vienādu nosaukumu – kārtējā funkcija pārraksta iepriekšējo, un atmiņā ir tikai viena, pēdējā

```
def a():
    print("pirmā")

def a(x):
    print("otrā")

a()
```

```
Traceback (most recent call last):
  File "C:/src/fun.py", line 7, in <module>
    a()
builtins.TypeError: a() missing 1 required positional
                    argument: 'x'
```

Att. 12.23. Funkcijas ar vienādu nosaukumu – pēdējā (tātad, vienīgā) funkcija ir ar vienu parametru, tāpēc ir kļūda, mēģinot izsaukt bez parametriem

12.3.4. Funkcijas ar mainīgu parametru skaitu

Iepriekš tika aprakstīti funkciju parametri, caur kuriem varēja padot argumentus pozicionāli vai pēc nosaukuma (sk. sadaļu 12.2.2), kas nozīmē, ka visiem izmantotajiem parametriem ir jābūt **tieši** aprakstītiem funkcijas definīcijā.

Tomēr *Python* ir iespēja norādīt funkciju parametrus **netieši**, un to iespējams organizēt divos veidos:

- parametru saraksts (tiks padoti argumenti bez nosaukumiem, apzīmē ar *),
- parametru vārdnīca (tiks padoti argumenti ar nosaukumiem, apzīmē ar **).

Parametru sarakstu apzīmē viens speciālais parametrs, pirms kura nosaukuma ir viena zvaigznīte *, un funkcija caur to padotos argumentus saņem kā slēgto sarakstu (*tuple*).

```
def printall(*aa):
    for a in aa: # pārlasa padotā saraksta elementus, t.i.,
                 argumentu vērtības
        print(a)

printall(1,2,3,4)
```

```
1
2
3
4
```

Att. 12.24. Parametru saraksts **aa*

Parametru vārdnīcu apzīmē viens speciālais parametrs, pirms kura nosaukuma ir divas zvaigznītes **, un funkcija caur to padotos argumentus saņem kā vārdnīcu (*dict*, šī datu struktūra tiks aprakstīta nodaļā 14), t.i., saņem gan argumentu nosaukumus (kas ir vārdnīcas atslēgas), gan argumentu vērtības (vārdnīcas vērtības).

```
def printall(**dd):
    for k in dd: # pārlasa vārdnīcas atslēgas, t.i., argumentu
                 nosaukumus
        print(k,dd[k]) # argumenta vērtībai piekļūst ar []

printall(a=1,b=2,c=3,d=4)
```

```
a 1
b 2
c 3
d 4
```

Att. 12.25. Parametru vārdnīca **dd*

Dažādi parametru tipi funkcijās var tikt kombinēti, bet tādā gadījumā ir jāievēro dažādo parametru tipu secība. Vienkāršākā shēma nosaka šādu parametru tipu secību (Att. 12.26), no kuriem daži var tikt izlaisti:

- pozicionālie parametri (*a,b*),
- parametru saraksts (*ss*),
- neobligātie nosauktie parametri (*x,y*),
- parametru vārdnīca (*dd*).

```
def fun(a,b,*ss,x=1,y=2,**dd):
    print(a,b) # pozicionālie parametri
    for e in ss: # parametru saraksts
        print(e,end=' ')
    print()
    print(x,y) # neobligātie nosauktie parametri
    for k in dd: # parametru vārdnīca
        print((k,dd[k]),end=' ')
    print()

fun('alpha','beta',1,2,3,x=11,i=100,j=200)
print('************************')
fun(1,2,3,y=22,i=100,j=200)
print('************************')
fun(1,2,3,i=100,j=200)
print('************************')
fun('alpha','beta',x=11,y=22,i=100,j=200)
```

```
alpha beta
1 2 3
11 2
('i', 100) ('j', 200)
************************
1 2
3
1 22
('i', 100) ('j', 200)
************************
1 2
3
1 2
('i', 100) ('j', 200)
************************
alpha beta

11 22
('i', 100) ('j', 200)
```

Att. 12.26. Dažādu parametru tipu kombinēšana

13. Mainīgie un objekti, maināmie (*mutable*) un nemaināmie (*immutable*) datu tipi

13.1. Atmiņas piešķiršana objektiem un atmiņas atbrīvošana

Valodā *Python* mainīgie ir tikai norādes uz vērtībām, t.i., objektiem, un ir speciāls mehānisms, kā objektiem tiek piešķirta atmiņa un pēc tam, kad tā nav vajadzīga, atbrīvota.

Objekts valodā *Python* ir vērtība (vērtību, datu kopums, kas uzskatāms par vienotu veselumu), kas glabājas atmiņā.

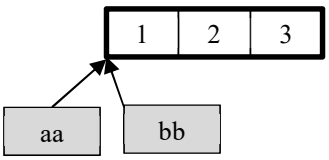
Objektam tiek **automātiski piešķirta atmiņa** brīdī, kad tiek veikta **piešķiršanas** operācija (mainīgajam).

Objekta lietotā atmiņa tiek **automātiski atbrīvota**, kad objekts vairāk "nav vajadzīgs", t.i., uz to nenorāda ("*nereferencējas*") neviens mainīgais (uz vienu un to pašu objektu var norādīt vairāki mainīgie).

Lai zinātu, kurā brīdī objekts vairs "nav vajadzīgs", katram objektam tiek veikta referenču uzskaite (*reference counting*), un kad referenču skaits uz objektu kļūst 0, tad objekts **vairs nav vajadzīgs**, un tā atmiņa var tikt atbrīvota.

```
aa = [1,2,3]
bb = aa
```

Att. 13.1. Divas references uz vienu objektu (sk. ilustrāciju Att. 13.2)



Att. 13.2. Divas references uz vienu objektu (atbilstoši kodam Att. 13.1)

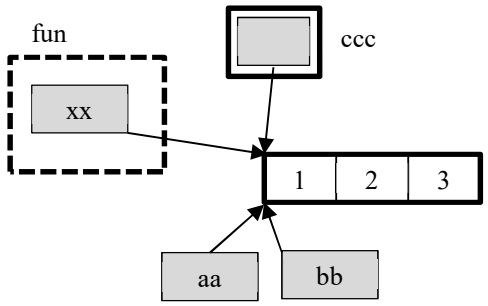
Lai zinātu, kurā brīdī uz objektu nenorāda neviena reference, ir jāzina darbības, kas uz objektu uzliek papildus references vai, tieši otrādi, tās noņem.

Objektam **references pievieno** šādas darbības:

- piešķiršanas operators,
- objekta nodošana caur parametru,
- objekta pievienošana sarakstam vai citai datu struktūrai.

```
def fun (xx):
    pass
aa = [1,2,3]
bb = aa
ccc = [aa]
fun(bb)
```

Att. 13.3. Vairākas references uz vienu objektu (sk. ilustrāciju Att. 13.4)



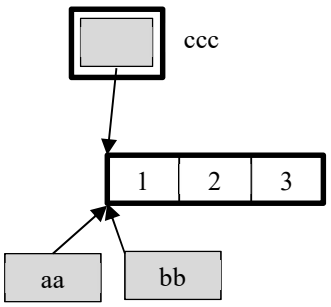
Att. 13.4. Vairākas (četras) references uz vienu objektu (stāvoklis funkcijas *fun* iekšienē atbilstoši kodam Att. 13.3)

Objektam **reference tiek atņemta** šādās situācijās:

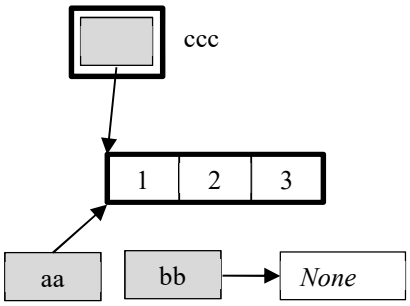
- mainīgajam tiek piešķirts cits objekts,
- programmas bloks, t.i., funkcija (kurā pievienota reference) beidzas,
- saraksts vai cita datu struktūra, kuram pievienots objekts, beidz eksistēt.

```
1. def fun (xx):
2.     print(sys.getrefcount(xx))
3. aa = [1,2,3]
4. bb = aa
5. ccc = [aa]
6. fun(bb)
7. bb = None
8. ccc = ["Hello","World"]
```

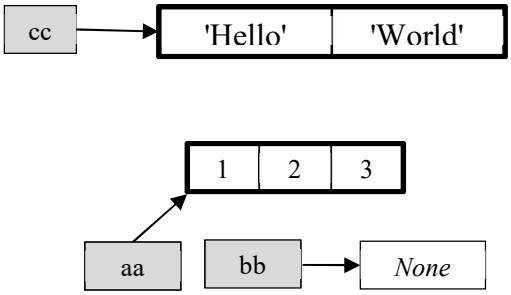
Att. 13.5. Vairākas references uz vienu objektu un to atņemšana (sk. ilustrācijas Att. 13.6, Att. 13.7, Att. 13.8)



Att. 13.6. Referenču atbrīvošana, uz sarakstu [1,2,3] palikušas 3 references (stāvoklis aiz funkcijas *fun* izsaukuma pēc rindas 6 atbilstoši kodam Att. 13.5, salīdzināt ar Att. 13.4)



Att. 13.7. Referenču atbrīvošana, uz sarakstu [1,2,3] palikušas 2 references (stāvoklis pēc rindas 7 atbilstoši kodam Att. 13.5, salīdzināt ar Att. 13.6)



Att. 13.8. Referenču atbrīvošana, uz sarakstu [1,2,3] palikusi 1 reference (stāvoklis pēc rindas 8 atbilstoši kodam Att. 13.5, salīdzināt ar Att. 13.7)

Operators *is* atgriež, vai abi mainīgie norāda uz **to pašu** objektu (atšķirībā no ==, kas atgriež, vai norāda uz **tādu pašu** objektu).

```
aa = [1,2,3]
bb = aa
cc = [1,2,3]
print(aa is bb, aa == bb) # tas pats objekts, tātad vienāds
print(aa is cc, aa == cc) # cits vienāds objekts
```

```
True True
False True
```

Att. 13.9. Operatori *is* un ==

13.2. Maināmie (mutable) un nemaināmie (immutable) datu tipi

Valodā *Python* ir divas svarīgas datu tipu kategorijas – maināmie (*mutable*) un nemaināmie (*immutable*). Konceptuāli nemaināmie datu tipi valodā *Python* ir ieviesti, lai objektiem (kas parasti ir ar relatīvi nelielu izmēru) nodrošinātu un garantētu *read-only* režīmu piekļuvē tiem, kas savukārt nozīmē potenciāli mazāk kļūdu saistībā ar nejaušu negribētu objekta mainīšanu.

Galvenie **nemaināmie** (*immutable*) datu tipi ir šādi (starp tiem ir visi galvenie pamattipi):

- *int* (vesels skaitlis),
- *bool* (loģiskais datu tips),
- *float* (skaitlis ar peldošo komatu),
- *str* (simbolu virkne),

- *tuple* (slēgtais saraksts).

Galvenie **maināmie** (*mutable*) datu tipi ir šādi:

- *list* (saraksts), vienīgais līdz šim apskatītais nemaināmais datu tips,
- *dict* (vārdnīca),
- *set* (kopa),
- lietotāja veidotās klases ar *class*.

Nemaināms (*immutable*) datu tips nozīmē, ka pēc izveidošanas objekts vairs nav maināms, un, ja tas it kā tiek darīts, tad patiesībā izveidojas cits objekts – pat ja vizuāli, no programmas koda viedokļa, tā izskatās kā izmaiņa.

```
aa = [1,2,3]
bb = aa
aa += [5,6]
print(aa)
print(bb, bb is aa) # aa un bb joprojām norāda uz to pašu
                    objektu

xx = "Hello"
yy = xx
xx += ", World!" # tika izveidots jauns objekts un xx tika
                    atvienots no vecā objekta
print(xx)
print(yy, yy is xx) # yy norāda uz veco objektu
```

```
[1, 2, 3, 5, 6]
[1, 2, 3, 5, 6] True
Hello, World!
Hello False
```

Att. 13.10. Operatora += pielietošana maināmam tipam (*list*) un nemaināmam tipam (*tuple*). Otrajā gadījumā tiek izveidots jauns objekts

Ja vienkāršākajā gadījumā nemaināmie datu tipi raksturojas ar to, ka vienmēr tiek veidots cits objekts, tad sarežģītākajā gadījumā objekta izmaiņa vienkārši netiek ļauta, un notiek kļūda.

```
s = "Hello"
s[0] = 'h'
```

```
Traceback (most recent call last):
  File "C:/src/hello.py", line 2, in <module>
    s[0] = 'h'
builtins.TypeError: 'str' object does not support item
                    assignment
```

Att. 13.11. Nemaināma (*immutable*) datu tipa (šeit: *str*) objektu nevar mainīt

Ja tomēr ir vēlēšanās "mainīt" nemaināma datu tipa objektu, tad patiesībā ir jāizveido cits, izmainīts objekts, un tas jāievieto vecā objekta vietā.

```
s = "Hello"
s = 'h' + s[1:]
print(s)
```

```
hello
```

Att. 13.12. Nemaināma (*immutable*) datu tipa objekta “maiņa”, pārrakstot ar jaunu vērtību

14. Vārdnīcas (*dict*) un citas ar atslēgas pieeju saistītas datu struktūras

14.1. Vārdnīca (dict)

Valodā *Python* viena no pamatstruktūrām ir *list* (saraksts), un programmēšanas valodās indeksējamas datu struktūras (saraksts, masīvs, vektors) ir vienas no biežāk lietotajām savas vienkāršās pielietošanas un saprotamības dēļ. Tomēr saraksti (masīvi) nav efektīvākā datu struktūra, lai piekļūtu elementiem pēc kaut kā **sarežģītāka nekā kārtas numurs**.

Saraksts valodā *Python* atbilst masīva jēdzienam citās programmēšanas valodās.

> **Vārdnīca** (*dictionary*, valodā *Python* – *dict*) ir datu struktūra, kurā elementi tiek identificēti pēc atslēgas (i).

Saturīgi ņemot, vārdnīca ir kā **tabula ar divām kolonnām** – atslēga un vērtība, kur katram ierakstam tiek nodrošināta efektīva pieeja pēc atslēgas (pēc līdzības ar parasto vārdnīcu, kur, pateicoties alfabētiskam sakārtojumam, var ātri atrast vajadzīgo šķirkli).

Vārdnīcas realizācijas pamatā ir sarežģīti algoritmi, kas ļauj efektīvi pārveidot atslēgu par attiecīgā ieraksta adresi (atrašanās vietu) vārdnīcā.

Tab. 14.1.

Vārdnīca kā tabula, sakārtota pēc atslēgas

atslēga (*key*)	vērtība (*value*)
četri	(4, 'four')
divi	(2, 'two')
trīs	(3, 'three')
viens	(1, 'one')

Atšķirībā no indeksa (jeb kārtas numura) masīvos un sarakstos, atslēgai ir šādas īpašības, kas paplašina tās pielietojumu:

- atslēga var būt sarežģītāks objekts nekā skaitlisks indekss (tipisks piemērs ir simbolu virkne, bet var būt arī, piemēram, slēgtais saraksts (*tuple*)),
- (kas lielā mērā izriet no pirmā punkta) struktūras objektu atslēgām nav jābūt pēc kārtas.

Tādējādi, vienu vārdnīcas elementu identificē:

- vārdnīcas vārds,
- atslēga.

14.1.1. Vārdnīcas izveidošana

Izšķir šādus vārdnīcas izveidošanas veidus:

1. tukšas vārdnīcas izveidošana ({}, *dict*()),
2. vārdnīcas izveidošana ar inicializācijas virkni {...},
3. vārdnīcas izveidošana ar atslēgu virkni, izmantojot metodi *fromkeys()*,
4. vārdnīcas izveidošana, kopējot citas vārdnīcas saturu ar *copy*() vai *deepcopy*(·), tāpat kā saraksta gadījumā (sk. sadaļu 9.1.3).

Tukšu vārdnīcu izveido vai nu, izmantojot funkciju *dict*, vai ar tukšām figūriekavām {}.

```
dd = {}
ee = dict()
print(dd, len(dd), type(dd))
print(ee, len(ee), type(ee))
```

```
{} 0 <class 'dict'>
{} 0 <class 'dict'>
```

Att. 14.1. Tukšas vārdnīcas izveidošana (funkcija *len* atgriež vārdnīcas lielumu)

Ar komatiem atdalītu pāru (*atslēga:vērtība*) virkne figūriekavās nozīmē vārdnīcas izveidošanu.

```
dd = {'četri':(4,'four'), 'divi':(2,'two'),
                  'trīs':(3,'three'), 'viens':(1,'one')}
print(dd)
print(len(dd))
print(list(dd.keys()))
```

```
{'četri': (4, 'four'), 'divi': (2, 'two'), 'trīs': (3,
                  'three'), 'viens': (1, 'one')}
4
['četri', 'divi', 'trīs', 'viens']
```

Att. 14.2. Vārdnīcas izveidošana ar inicializācijas virkni

Ar funkciju *fromkeys()* veidotai vārdnīcai tiek norādītas atslēgas, bet visas ierakstu vērtības tiek uzstādītas vienādas (ja nav norādītas, tad *None*).

```
kk = ('viens', 'divi', 'trīs')
dd = dict.fromkeys(kk) # vārdnīca, nenorādot ierakstu vērtības
print(dd)
val = 0
dd = dict.fromkeys(kk,val) # vārdnīca, nosakot ierakstu
                  vērtības
print(dd)
```

```
{'viens': None, 'divi': None, 'trīs': None}
{'viens': 0, 'divi': 0, 'trīs': 0}
```

Att. 14.3. Vārdnīcas izveidošana ar atslēgu virkni un *fromkeys()*

14.1.2. Piekļūšana vārdnīcas ierakstiem

Piekļūšana vārdnīcas ierakstiem notiek pa vienam, izmantojot atslēgu ar tādu pašu sintaksi [], kā ar indeksu piekļūstot saraksta elementam.

```
dd = {'četri':(4,'four'), 'divi':(2,'two'),
                  'trīs':(3,'three'), 'viens':(1,'one')}
print(dd['divi'])
dd['divi'] = 'ZWEI'
print(dd)
```

```
(2, 'two')
```

```
{'četri': (4, 'four'), 'divi': 'ZWEI', 'trīs': (3, 'three'),
                    'viens': (1, 'one')}
```

Att. 14.4. Piekļūšana vārdnīcas ieraksta vērtībai lasīšanas un rakstīšanas režīmos

Python kontrolē atslēgas eksistenci vārdnīcā.

```
dd = {'četri':(4,'four'), 'divi':(2,'two'),
                    'trīs':(3,'three'), 'viens':(1,'one')}
print(dd['pieci'])
```

```
Traceback (most recent call last):
  File "C:/src/dict.py", line 2, in <module>
    print(dd['pieci'])
builtins.KeyError: 'pieci'
```

Att. 14.5. *Python* kļūdas paziņojums pie neeksistējošas atslēgas

14.1.3. Vārdnīcas elementu pārlasīšana un piederības pārbaude

Saraksta elementu pārlasīšana var notikt šādos veidos (sk. arī nodaļu 7.1):

- atslēgu iterācija – *for k in d.keys()...*,
- ierakstu iterācija – *for i in d.items()...*

Visvienkāršākais veids ir pārlasīt visas atslēgas, izmantojot funkciju *keys()* un tad vajadzības gadījumā pēc tās piekļūt vērtībai.

```
dd = {'četri':(4,'four'), 'divi':(2,'two'),
                    'trīs':(3,'three'), 'viens':(1,'one')}
for k in dd.keys():
    print(k,dd[k])
```

```
četri (4, 'four')
divi (2, 'two')
trīs (3, 'three')
viens (1, 'one')
```

Att. 14.6. Vārdnīcas ierakstu pārlasīšana ar atslēgu iterāciju

Ierakstu iterācijas (izmantojot funkciju *items()*) elements ir pāris (atslēga: vērtība) slēgtā saraksta formā.

```
dd = {'četri':(4,'four'), 'divi':(2,'two'),
                    'trīs':(3,'three'), 'viens':(1,'one')}
for i in dd.items():
    print(i,i[0],i[1])
```

```
('četri', (4, 'four')) četri (4, 'four')
('divi', (2, 'two')) divi (2, 'two')
('trīs', (3, 'three')) trīs (3, 'three')
('viens', (1, 'one')) viens (1, 'one')
```

Att. 14.7. Vārdnīcas ierakstu pārlasīšana ar ierakstu iterāciju

Vārdnīcas ieraksta piederības pārbaude pēc atslēgas notiek, izmantojot *in* un *not in* operatorus.

```
dd = {'četri':(4,'four'), 'divi':(2,'two'),
                  'trīs':(3,'three'), 'viens':(1,'one')}
print('trīs' in dd, 'trīs' not in dd)
print('pieci' in dd, 'pieci' not in dd)
```

```
True False
False True
```

Att. 14.8. *in* un *not in* atslēgas piederības pārbaudei vārdnīcā

14.1.4. Ierakstu pievienošana un izdzēšana no vārdnīcas

Viena ieraksta pievienošana vārdnīcai tiek veikta ar operatoru []:

```
dd = {}
dd[3]='three'
dd[5]='five'
print(dd)
```

```
{3: 'three', 5: 'five'}
```

Att. 14.9. Ierakstu pievienošana vārdnīcai

Ieraksta izdzēšanai no vārdnīcas lieto operatoru *del* vai metodi *pop*.

```
dd = {'četri':(4,'four'), 'divi':(2,'two'),
                  'trīs':(3,'three'), 'viens':(1,'one')}
del dd['divi']
dd.pop('trīs')
print(dd)
```

```
{'četri': (4, 'four'), 'viens': (1, 'one')}
```

Att. 14.10. Vārdnīcas ierakstu izdzēšana ar *del* un *pop*

14.2. Kopa (set)

Kopa ir unikālu objektu kopums.

No vārdnīcas (*dict*) skatupunkta raugoties, kopa (*set*) ir tāda vārdnīca, kurā glabājas tikai atslēgas, tomēr ar papildus funkcionalitāti.

14.2.1. Kopas izveidošana

Izšķir šādus kopas izveidošanas veidus:

1. tukšas kopas izveidošana (*set*()),
2. kopas izveidošana ar inicializācijas virkni {...},

Tukšu kopu izveido, izmantojot funkciju *set,* skat. Att.14.11.

```
ss = set()
print(list(ss), len(ss), type(ss))
```

```
[] 0 <class 'set'>
```

Att. 14.11. Tukšas kopas izveidošana (funkcija *len* atgriež kopas lielumu)

Ar komatiem atdalītu vērtību virkne figūriekavās nozīmē kopas izveidošanu.

```
ss = {1,3,2,4}
print(ss)
```

```
{1, 2, 3, 4}
```

Att. 14.12. Kopas izveidošana ar inicializācijas virkni

14.2.2. Piekļūšana kopas elementiem un piederības pārbaude

Kolīdz elements pievienots kopai, to vairs nevar mainīt; var elementu izdzēst, var pievienot citus elementus.

Vienīgais veids kā piekļūt kopas elementiem, ir tās elementu pārlase.

```
ss = {'viens','divi','trīs'}
for s in ss:
    print(s)
```

```
viens
trīs
divi
```

Att. 14.13. Kopas elementu pārlasīšana

Kopas elementa piederības pārbaude notiek, izmantojot *in* un *not in* operatorus.

```
ss = {'viens','divi','trīs'}
print('trīs' in ss, 'trīs' not in ss)
print('pieci' in ss, 'pieci' not in ss)
```

```
True False
False True
```

Att. 14.14. *in* un *not in* elementa piederības pārbaudei kopā

14.2.3. Elementu pievienošana un izdzēšana no kopas

Viena elementa pievienošana kopai tiek veikta ar funkciju *add()*:

```
ss = set()
ss.add(22)
ss.add(44)
ss.add(33)
print(ss)
```

```
{33, 44, 22}
```

Att. 14.15. Elementu pievienošana kopai pa vienam

Vairāku elementu pievienošana kopai tiek veikta ar funkciju *update()*:

```
ss = {99}
ss.update([44,55,66,77])
print(ss)
```

```
{66, 99, 44, 77, 55}
```

Att. 14.16. Vairāku elementu pievienošana kopai

Elementa izdzēšanai no kopas lieto metodi *remove* vai *discard* (ja elements neeksistē, tad *remove* izsauc kļūdu, bet *discard* nē, citādi metodes darbojas vienādi)

```
ss = {'viens','divi','trīs'}
ss.remove('viens')
ss.discard('trīs')
print(ss)
```

```
{'trīs', 'divi'}
```

Att. 14.17. Kopas ierakstu dzēšana ar *remove* un *discard*

14.2.4. Kopas specifiskās darbības – apvienošana un šķēlums

Divu kopu apvienošanu iegūst ar metodi *union()*:

```
ss = {1,2,3,4}
tt = {3,4,5,6}
uu = tt.union(ss)
print(ss)
print(tt)
print(uu)
```

```
{1, 2, 3, 4}
{3, 4, 5, 6}
{1, 2, 3, 4, 5, 6}
```

Att. 14.18. Metode *union* kopu apvienošanai

Divu kopu šķēlumu iegūst ar metodi *intersection()*:

```
ss = {1,2,3,4}
tt = {3,4,5,6}
uu = tt.intersection(ss)
print(ss)
print(tt)
print(uu)
```

```
{1, 2, 3, 4}
{3, 4, 5, 6}
{3, 4}
```

Att. 14.19. Metode *intersection* kopu šķēluma iegūšanai

14.3. Specializētā vārdnīca Counter

Standarta vārdnīca *dict*, ja pie tās vēršas ar neeksistējošu atslēgu, izsauc kļūdu. Dažu uzdevumu risināšanā piemēram, skaitīšanā, tas rada neērtības, tāpēc bibliotēkā *collections* blakus citām datu struktūrām ir pieejama specializēta kopa *Counter*.

Vārdnīca *Counter* ir ērta elementu uzskaitei – vēršoties šajā vārdnīcā pēc neeksistējošas atslēgas, tiek automātiski atgriezta vērtība *0*, tāpēc atslēgas eksistence nav jāpārbauda.

```
from collections import Counter
aa = ['a','b','c','b','c','d','b']
cc = Counter()
for a in aa:
    cc[a] += 1
for k in cc:
    print(k,cc[k])
```

```
a 1
b 3
c 2
d 1
```

Att. 14.20. Klases *Counter* pielietošana skaitīšanai

Ja tas pats būtu jāizdara ar parasto *dict*, būtu jāpievieno divas rindas neeksistējošas atslēgas kontrolei (Att. 14.21):

```
aa = ['a','b','c','b','c','d','b']
cc = {} # parastā vārdnīca
for a in aa:
    if a not in cc: # pārbaude uz eksistenci
        cc[a] = 0
    cc[a] += 1
for k in cc:
    print(k,cc[k])
```

```
a 1
b 3
c 2
d 1
```

Att. 14.21. Skaitīšana ar parasto vārdnīcu, nevis *Counter*

```
from collections import Counter
cc = Counter()
dd = {}
print(cc['ok']) # atgriež 0
print(dd['ok']) # atgriež kļūdu
```

```
0Traceback (most recent call last):
  File "c:/src/counter.py", line 5, in <module>
    print(dd['ok'])
builtins.KeyError: 'ok'
```

Att. 14.22. Piekļūšana neeksistējošai atslēgai ar parasto vārdnīcu un *Counter*

Metodes *sorted* izmantošana rezultātu sakārtošanai, vispirms parādot biežākos.

```
from collections import Counter
aa = ['a','b','c','b','c','d','b']
cc = Counter()
for a in aa:
    cc[a] += 1
for k in sorted(cc, key=lambda x: cc[x], reverse=True):
    print(k,cc[k])
```

```
b 3
c 2
a 1
d 1
```

Att. 14.23. Klases *Counter* pielietošana skaitīšanai ar sakārtošanu

15. Funkciju un datu struktūru speciālās tēmas

15.1. Lambda funkcijas

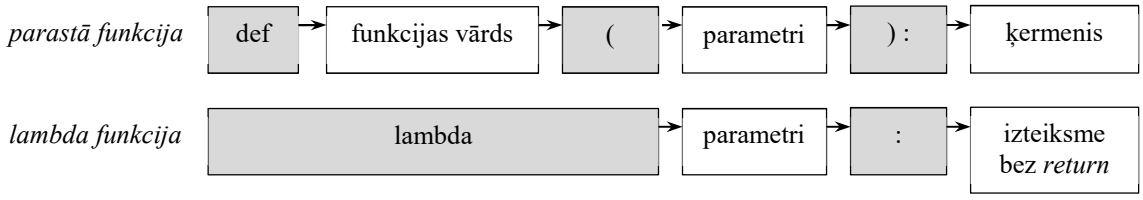
Lambda funkcija ir maza anonīma funkcija, tātad:

- tai nav vārda,
- tās ķermenī ir tikai viena izteiksme (atgriež vienu vērtību, un nevajag *return*).

Kaut arī lambda funkcijas var lietot vienkārši kā aizvietotāju parastajām (Att. 15.2), tādējādi "taupot" nosaukumus, tomēr parasti tās izmanto kontekstos, kad vajadzīga funkciju izsaukšana netieši ar automātisku argumentu padošanu tām kāda lielāka procesa ietvaros. Tipiskākais piemērs ir datu kārtošana ar *sort* un *sorted* (sk. sadaļu 15.2.2).

No izmantošanas viedokļa:

- lambda funkciju definē tās izsaukšanas vietā,
- lambda funkciju nevis vienkārši izsauc, bet izsaucot to piešķir, atgriež vai (tieši vai netieši) padod kā argumentu.



Att. 15.1. Parastās un lambda funkcijas uzbūves shēma

```
def parasta_summa (a,b):
    return a+b

# lambda funkciju piešķir mainīgajam:
lsumma = lambda a,b: a+b

print(parasta_summa(5,7)) # izsauc parasto funkciju
print(lsumma(5,7)) # izsauc lambda funkciju
```

```
12
12
```

Att. 15.2. Lambda funkcija kā parastas funkcijas alternatīva

Valodā *Python* funkciju var padot caur parametru, atgriezt, kā arī veidot funkciju funkcijā, nākošais piemērs to demonstrē, iesaistot lambda funkciju (Att. 15.3).

```
def mana_summa(n):
  return lambda a : a + n

print(mana_summa(7)) # atgriež lambda funkciju
print(mana_summa(7)(5)) # atgriež rezultātu
mana_summa2 = mana_summa(7)
print(mana_summa2(5)) # rezultātu var atgriezt arī tā
```

```
<function mana_summa.<locals>.<lambda> at 0x02E69970>
12
12
```

Att. 15.3. Lambda funkcija citas funkcijas sastāvā

Tomēr īsti jēgpilns lambda funkcijas lietojums parādīts sadaļā par kārtošanu (15.2.2).

15.2. Datu kārtošana ar sort un sorted

15.2.1. Vienkārša kārtošana

Metodes *sort* un *sorted* ir paredzētas datu kārtošanai un abas nodrošina līdzīgu funkcionalitāti, tikai:

- *sort* kārto "uz vietas" (*on place*), t.i. maina pašu datu struktūru, neveidojot jaunu un ir pieejama tikai sarakstam *list*,
- *sorted* saglabā oriģinālo datu struktūru un izveido jaunu sakārtotu struktūru.

To izmantošana nedaudz atšķiras arī sintaktiski:

- Metode *sort* tiek izsaukta kā saraksta metode:
  `s.sort()`
- Metode *sorted* ir neatkarīga funkcija:
  `sorted(s)`

```
aa = ["one","two","three","four"]
aa2 = sorted(aa) # taisa sakārtotu kopiju
print(aa) # nesakārtots
print(aa2)
aa.sort() # kārto uz vietas
print(aa)
```

```
['one', 'two', 'three', 'four']
['four', 'one', 'three', 'two']
['four', 'one', 'three', 'two']
```

Att. 15.4. Vienkāršs kārtošanas piemērs

Lai sakārtotu pretējā secībā, kārtošanas metodēm jānorāda parametrs *reverse* un jāuzstāda uz *True* (pēc noklusējuma tas ir *False*).

```
aa = ["one","two","three","four"]
aa2 = sorted(aa,reverse=True) # taisa sakārtotu kopiju
print(aa) # nesakārtots
print(aa2)
aa.sort(reverse=True) # kārto uz vietas
print(aa)
```

```
['one', 'two', 'three', 'four']
['two', 'three', 'one', 'four']
['two', 'three', 'one', 'four']
```

Att. 15.5. Apgriezta (reversa) kārtošana

15.2.2. Kārtošana pēc atslēgas

Atslēga ir kārtojamā objekta reprezentācija kārtošanas vajadzībām, kas pēc noklusējuma ir pats objekts.

Ja nepieciešama speciāla kārtošana, tad ir nepieciešama funkcija (ar vienu parametru), kas no kārtojamā objekta izveido atslēgu, un šī funkcija tad ir jāpadod kārtošanas metodei ar parametru *key*.

Kārtošanas atslēgas veidošanas funkciju:

- var paņemt gatavu (piemēram, *int*, *str*, *len*, Att. 15.6),
- var izveidot savu –
    - kā parastu funkciju (Att. 15.7),
    - kā lambda funkciju – šī ir tipiska lambda funkciju izmantošanas vieta (Att. 15.8).

```
aa = ["one","two","three","four"]
aa2 = sorted(aa,key=len) # pēc garuma augoši
aa3 = sorted(aa,key=len,reverse=True) # pēc garuma dilstoši
print(aa2)
print(aa3)
```

```
['one', 'two', 'four', 'three']
['three', 'four', 'one', 'two']
```

Att. 15.6. Kārtošana pēc garuma ar iebūvēto funkciju *len*

Kārtošanu var veikt sarežģītāk, un tad jāveido speciāla viena parametra metode, kas atbilstoši pārveido objektu par atslēgu.

Nākošajā piemērā sakārtojums notiks alfabētiski pēc apgriezta vārda, apgriešanu veicot funkcijai *myrev*.

```
def myrev(s):
    return s[::-1]

aa = ["one","two","three","four"]
print([myrev(s) for s in aa]) # funkcijas myrev pārbaude

aa2 = sorted(aa,key=myrev) # sakārto pēc apgriezta vārda
print(aa2)
```

```
['eno', 'owt', 'eerht', 'ruof']
['three', 'one', 'two', 'four']
```

Att. 15.7. Kārtošana pēc apgriezta vārda ar pašizveidotu funkciju

Lai neveidotu nosauktu funkciju, tās vietā šeit ērti (t.sk. īsāk) izmantot lambda funkciju.

```
aa = ["one","two","three","four"]

aa2 = sorted(aa,key=lambda s: s[::-1])
print(aa2)
```

```
['three', 'one', 'two', 'four']
```

Att. 15.8. Kārtošana pēc apgriezta vārda ar lambda funkciju

Vienu un to pašu datu struktūru var kārtot dažādos veidos.

```
dd = {'četri':(4,'four'), 'divi':(2,'two'),
                  'trīs':(3,'three'), 'viens':(1,'one')}
print("== izdruka pēc atslēgas (latviskā nosaukuma) ==")
for k in dd:
    print(k,dd[k])
print("== izdruka, kārtojot pēc vērtības ==")
# dd padod atslegu sarakstu, tas pats, kas dd.keys(),
# x - viena konkrētā padotā atslēga procesa laikā
# dd[x] - vērtība pēc konkrētās atslēgas
# [0] - šeit: vērtības 0. pozīcija
for k in sorted(dd,key=lambda x: dd[x][0]):
    print(k,dd[k])
print("== izdruka, kārtojot pēc angliskā nosaukuma ==")
# [1] - šeit: vērtības 1. pozīcija
for k in sorted(dd,key=lambda x: dd[x][1]):
    print(k,dd[k])
```

```
== izdruka pēc atslēgas (latviskā nosaukuma) ==
četri (4, 'four')
divi (2, 'two')
trīs (3, 'three')
viens (1, 'one')
== izdruka, kārtojot pēc vērtības ==
viens (1, 'one')
divi (2, 'two')
trīs (3, 'three')
četri (4, 'four')
== izdruka, kārtojot pēc angliskā nosaukuma ==
četri (4, 'four')
viens (1, 'one')
trīs (3, 'three')
divi (2, 'two')
```

Att. 15.9. Vārdnīcas izdruka dažādos sakārtojumos

15.2.3. Alfabētiska kārtošana latviešu valodā

Alfabētiskā kārtošana latviešu valodā ir salīdzinoši sarežģīta, jo attiecīgā burta vieta kārtojumā nav neatkarīga, bet atkarīga no konteksta – un tas attiecas uz burtiem ar diakritiskajām zīmēm, piemēram, ‘ā’, ‘ņ’, ‘š’.

Parastais nosacījums ir tāds, ka ‘ā’ tiek uzskatīts par līdzvērtīgu ‘a’, tomēr, ja divi vārdi atšķiras tikai pēc šī burta (‘ā’ vai ‘a’), tad gan ‘ā’ tiek kārtots aiz ‘a’.

Tab. 15.1.

Latviešu valodas teksta vienību alfabētiska kārtošana

Oficiālais kārtojums latviešu valodā	**Windows 10 (Word, Excel) noklusētais kārtojums (‘ā’ vienmēr aiz ‘a’)**	**Python noklusētais kārtojums (‘ā’ vienmēr aiz ‘a’ un lielie burti priekšā)**
auglis	auglis	Sala
sākums	sala	Zilonis
sala	Sala	auglis
Sala	sals	sala
sals	sākums	sals
sāls	sāls	sākums
Zilonis	Zilonis	sāls

Tehniski, lai to nodrošinātu, būtu jābūvē divdaļīga kārtošanas atslēga:

- primārā daļa (diakritiskās zīmes tiek noņemtas, burtu reģistrs noņemts),
- sekundārā daļa nozīmīguma secībā –
    - diakritiskās zīmes kodējums,
    - burta reģistra kodējums (liksim mazos burtus pirms lielajiem).

Šim nolūkam ir jāizveido tabula speciālo latviešu burtu sasaistei ar atbilstošajiem latīņu burtiem (Att. 15.10).

```
lv = {'ā':'a', 'č':'c', 'ē':'e', 'ģ':'g', 'ī':'i',
      'ķ':'k', 'ļ':'l', 'ņ':'n', 'ō':'o', 'ŗ':'r',
      'š':'s', 'ū':'u', 'ž':'z'}
def lvkey(text):
    key1 = ""
    key2 = ""
    for c in text:
        k = 0
        if c.isupper():
            k += 1
            c = c.lower()
        if c in lv:
            k += 2
            c = lv[c]
        key1 += c
        key2 += str(k)
    return key1,key2

aa = ['sākums','Zilonis','sala','sāls','auglis','Sala','sals']
for a in sorted(aa,key=lvkey):
    print(a)
```

```
auglis
sākums
sala
Sala
sals
sāls
Zilonis
```

Att. 15.10. Kārtošana latviešu valodā, izmantojot funkciju

Tā kā atslēgas izveidošanas funkcijai nepieciešami dati, tad kārtošanas funkcionalitātes realizācijai labāk izmantot klasi (Att. 15.11) (par klasēm sk. nodaļu 16, bet klases līmeņa elementiem, kas šeit lietoti, sadaļu 16.4.2).

```
class lvsort:
    lv = {'ā':'a', 'č':'c', 'ē':'e', 'ģ':'g', 'ī':'i',
          'ķ':'k', 'ļ':'l', 'ņ':'n', 'ō':'o', 'ŗ':'r',
          'š':'s', 'ū':'u', 'ž':'z'}
    @classmethod
    def lvkey(cls,text):
        key1 = ""
        key2 = ""
        for c in text:
            k = 0
            if c.isupper():
                k += 1
                c = c.lower()
            if c in cls.lv:
                k += 2
                c = cls.lv[c]
            key1 += c
            key2 += str(k)
        return key1,key2

aa = ['sākums','Zilonis','sala','sāls','auglis','Sala','sals']
for a in sorted(aa,key=lvsort.lvkey):
    print(a)
```

```
auglis
sākums
sala
Sala
sals
sāls
Zilonis
```

Att. 15.11. Kārtošana latviešu valodā, izmantojot klasi

15.3. Funkcijas-ģeneratori

Ģenerators ir konstrukcija, kas katrā solī, pie tās vēršoties, atgriež **nākošo vērtību**. Tas ir ērts veids, kā pārlasīt dažādas vērtības.

No informācijas satura viedokļa ir divi ģeneratoru veidi:

- tie, kas pārvietojas pa esošās datu struktūrās **saglabātiem datiem** (ikviena iterējoša datu struktūra no datu pārlasīšanas viedokļa ir ģenerators – *list*, *tuple*, *str*, kā arī pašveidotu klašu objekti, kā aprakstīts sadaļā 16.6.3),
- tie, kas katrreiz **izrēķina nākošo vērtību**, piemēram, range, un kas var tikt veidoti, izmantojot **funkcijas-ģeneratorus**, kas tiks apskatīti šajā nodaļā, vai arī iterējamas datu struktūras (kā aprakstīts sadaļā 16.6.3).

Tehniski valodā *Python* funkcijas-ģeneratori ir tādas specifiskas funkcijas (Att. 15.12),

- kuras izsaucot nevis izpildās līdz galam un beidzas, bet gan tā vietā apstājas izpildes vidū, atgriežot vērtību, bet nākošajā izsaukumā turpina no tās vietas, kur pirms tam apstājas,
- kurām, lai to nodrošinātu, nosacīti ir sava iekšējā atmiņa, kas atceras apstāšanās pozīciju izpildē,
- kurām attiecīgi *return* vietā raksta ***yield***.

```
def simple_range(a,b):
    i = a
    while i<b:
        yield i
        i += 1

for i in simple_range(10,15):
    print(i)
```

```
10
11
12
13
14
```

Att. 15.12. Vienkāršs intervāla pārstaigāšanas ģenerators

Funkcijas-ģeneratora cikls drīkst arī "neapstāties", bet tad to nedrīkst darbināt ar *for* ciklu, bet gan tikai manuāli, izsaucot metodi *__next__* un apstāšanos nodrošināt ārēji (Att. 15.13).

```
def month_range(s=0):
    aa =
                ['Jan','Feb','Mar','Apr','May','Jun','Jul','A
                ug','Sep','Oct','Nov','Dec']
    while True:
        yield aa[s]
        s = (s + 1) % 12

month_range_from_october = month_range(9)
for i in range(5):
    print(month_range_from_october.__next__())
```

```
Oct
Nov
Dec
Jan
Feb
```

Att. 15.13. Mēnešu pārlasīšanas ģenerators ar mūžīgo ciklu

16. Objektorientētā programmēšana

16.1. Klase kā objektu īpašību apraksts

Programmēšanas valoda ir viens no algoritma pieraksta veidiem. Algoritms ir soļu, t.i., darbību virkne. Kaut arī programma sastāv gan no **darbībām**, gan **datiem**, tomēr klasiskais programmu strukturēšanas veids ir tieši pēc darbībām, un viens no galvenajiem programmas strukturēšanas elementiem ir funkcijas. Tādējādi tradicionāli no strukturēšanas viedokļa darbības ir primāras, bet dati sekundāri.

Tomēr lielā daļā reālās pasaules problēmu tieši **datiem ir primāra nozīme**. Objektorientētā programmēšana nodrošina šo principu programmēšanas līmenī, panākot, ka programma vispirms tiek strukturēta pēc datiem un tikai tad pēc darbībām.

> **Objektorientētā programmēšana** (*object-oriented programming*) ir programmēšanas paradigma, kuru raksturo tas, ka dati un darbības tiek apvienoti kopējās struktūrās, kur dati ieņem primāro lomu.

Tehniski objektorientēto programmēšanu reprezentē klases un objekti.

> **Objekts** (*object*) programmā ir datu kopums kopā ar tam piesaistītajām darbībām. Datus objektā reprezentē **lauki** (*fields, properties*) jeb **iekšējie mainīgie** (*member variables*), bet darbības reprezentē **metodes** (*methods*) jeb **iekšējās funkcijas** (*member functions*).

Turpmāk attiecīgi termini "lauki" un "iekšējie mainīgie", kā arī "metodes" un "iekšējās funkcijas" tiks lietoti kā sinonīmi.

Lai varētu izmantot objektus, vispirms ir jādefinē klase, kas ir viena tipa objektu īpašību apraksts.

> **Klase** (*class*) ir objekta tipa apraksts, kas ietver sevī iekšējo mainīgo deklarēšanu un iekšējo funkciju aprakstu.

Lielā daļā programmēšanas valodu klase tā arī tiek definēta – kā **lauku un iekšējo funkciju kopums**, bet visiem attiecīgās klases objektiem automātiski tiek izdalīta vienāda atmiņa – tā, kas vajadzīga lauku vērtību glabāšanai.

Tā kā valodā *Python* mainīgajiem tiek izdalīta atmiņa nevis speciālas deklarēšanas laikā, bet gan, piešķirot vērtību, tad klase ir **iekšējo funkciju kopums**, bet lauki tiek veidoti programmas kodā, veicot piešķiršanu (parasti tas notiek kādā iekšējā funkcijā, bet var notikt arī citur). Tādējādi teorētiski dažādiem vienas klases objektiem var būt dažāds lauku komplekts.

Līdzīgi kā parasto funkciju gadījumā, klasē nevar būt vairākas metodes ar vienu nosaukumu.

Python klases veidošanas specifika:

- klasi iesāk atslēgas vārds *class* (tas ir līdzīgi citām valodām),
- klasē tiek uzrādītas **tikai metodes**, bet netiek tieši definēti lauki (par klases līmeņa laukiem un metodēm sk. zemāk),

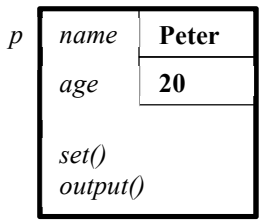
- katras metodes **pirmais parametrs** ir norāde uz attiecīgo objektu (tradicionāli šo parametru sauc *self*, lai arī tā to nosaukt nav obligāti)
    - tātad katrai metodei ir vismaz viens parametrs,
    - izsaucot metodi **caur objektu**, pirmais parametrs nav jānorāda (Att. 16.1),
    - izsaucot metodi **caur klasi** (tā parasti nedara, jo tas nav īsti objektorientēti), pirmajā parametrā tieši jāievieto objekts (Att. 16.3),
- piekļūšana objekta elementiem (laukiem un metodēm) notiek, izmantojot punkta notāciju (.).

```
class person:
    def set(self,name,age): # metodes definēšana
        self.name = name # šeit tiek izveidots lauks name
        self.age = age # šeit tiek izveidots lauks age
    def output(self): # metodes definēšana
        print(self.name,self.age)

p = person() # objekta izveidošana
p.set("Peter",20)
p.output()
```

```
Peter 20
```

Att. 16.1. Vienkārša klase ar diviem laukiem (pareizāk gan būtu teikt, ka objektam p ir divi lauki) un divām metodēm (atbilstoši ilustrācijai Att. 16.2)



Att. 16.2. Klases objekts atbilstoši programmai Att. 16.1

```
class person:
    def set(self,name,age): # metodes definēšana
        self.name = name # šeit tiek izveidots lauks name
        self.age = age # šeit tiek izveidots lauks age
    def output(self): # metodes definēšana
        print(self.name,self.age)

p = person() # objekta izveidošana
person.set(p,"Peter",20)
person.output(p)
```

```
Peter 20
```

Att. 16.3. Metožu izsaukšana caur klasi (objekts tieši jāliek pirmajā parametrā) – tā pati funkcionalitāte, kas Att. 16.1

Šo pašu funkcionalitāti (Att. 16.1) var sasniegt arī, izveidojot "tukšu" klasi, kuras objekti tad ir tikai datu struktūras bez darbībām (Att. 16.4).

```
class person:
    pass # klasē nekā nav

p = person() # objekta izveidošana
p.name = "Peter" # lauka name izveidošana no ārpuses
p.age = 20 # lauka age izveidošana no ārpuses
print(p.name,p.age)
```

```
Peter 20
```

Att. 16.4. Tukša klase un tās izmantošana (joprojām atbilstoši ilustrācijai Att. 16.2 no datu viedokļa)

16.2. Inkapsulācija – objekta elementu slēpšana

16.2.1. Inkapsulācijas pamatprincipi

Inkapsulācija (*encapsulation*) ir princips, kas objekta līmenī nodrošina slēpšanu, t.i., neļauj piekļūt "no ārpuses", kas objekta gadījumā nozīmē, ka elementam var piekļūt tikai caur kādu no metodēm. Parasto funkciju gadījumā šādu slēpšanu nodrošina procedurālās abstrakcijas princips, kas nozīmē, ka no ārpuses funkcijas iekšējiem mainīgajiem un citiem elementiem nav iespējams piekļūt, bet tikai izmantojot parametrus un atgriešanas mehānismu.

Iepriekšējā piemērā (Att. 16.4) redzams, ka laukiem *name* un *age* tiek piekļūts no ārpuses – tātad tie nav "inkapsulēti". Inkapsulētam (paslēptam) elementam nebūtu bijis iespējas piekļūt, izmantojot notāciju *objekts.elements*, bet tikai netieši caur citu metodi, kas nav paslēpta.

Citās programmēšanas valodās objekta elementu pieejamības regulēšanai tiek lietoti elementu modifikatori *private* (nav piekļuves no ārpuses) un *public* (ir piekļuve no ārpuses), kā arī viens vai vairāki papildus modifikatori pieejas detalizētākai noteikšanai (piemēram, *protected* klašu mantošanas gadījumā).

Slēpto (*private*) un atklāto (*public*) elementu nošķiršana klasē ir svarīga, lai nodalītu klases interfeisu (no ārpuses pieejamo funkcionalitāti) no iekšējās funkcionalitātes.

16.2.2. Slēpto (*private*) elementu definēšana

Praktiski **inkapsulācija** ir slēpto elementu definēšana, un parasti tas tiek veikts ar speciālu modifikatoru *private* un nozīmē, ka no ārpuses šim elementam piekļūt nevar principā.

Valodā *Python* šāda mehānisma **nav** – nav aizliegts piekļūt nevienam elementam no ārpuses, tā vietā piekļuve "it kā" slēptajiem elementiem tiek vienkārši padarīta **neērtāka** ("samudžinot" to nosaukumus), līdz ar to tos varētu saukt arī par **pseido-slēptajiem** elementiem.

Slēptos elementus valodā *Python* definē caur to nosaukumu – kas sākas ar **divām pasvītrojuma zīmēm** (__), kas nozīmē, ka tiem:

- nevar piekļūt tieši ar *objekts.__elements* (Att. 16.5),
- bet var piekļūt specifiski ar *objekts._klase__elements* (Att. 16.6).

```
class person:
    def set(self,name,age): # metodes definēšana
        self.__name = name # šeit tiek izveidots slēptais
                    lauks name
        self.__age = age # šeit tiek izveidots slēptais lauks
                    age
    def output(self): # metodes definēšana
        print(self.__name,self.__age)

p = person() # objekta izveidošana
p.set("Peter",20)
print(p.__name) # nevar piekļūt slēptajam laukam, izmet kļūdu
p.output()
```

```
Traceback (most recent call last):
  File "C:/src/person.py", line 10, in <module>
    print(p.__name) # nevar piekļūt slēptajam laukam, izmet
                    kļūdu
builtins.AttributeError: 'person' object has no attribute
                    '__name'
```

Att. 16.5. Slēptajam elementam (nosaukums sākas ar __) nevar piekļūt tieši

```
class person:
    def set(self,name,age): # metodes definēšana
        self.__name = name # šeit tiek izveidots slēptais
                    lauks name
        self.__age = age # šeit tiek izveidots slēptais lauks
                    age
    def output(self): # metodes definēšana
        print(self.__name,self.__age)

p = person() # objekta izveidošana
p.set("Peter",20)
print(p._person__name) # var piekļūt slēptajam laukam šādi
p.output() # var piekļūt slēptajam laukam caur metodi
```

```
Peter
Peter 20
```

Att. 16.6. Piekļuve slēptajam laukam var notikt (a) tieši specifiski vai (b) netieši

Papildus nianses darbā ar slēptajiem laukiem:

- slēpto (*private*) elementu nosaukums sākas ar divām pasvītrojuma zīmēm (piemēram, *__mansprivatais*), bet slēptie elementi nav vienīgie šāda veida specifiskie elementi;
- vēl ir arī t.s. aizsargātie (*protected*) elementi, kuru nosaukums sākas ar vienu pasvītrojuma zīmi, piemēram, *_mansaizsargatais*, un šādi elementi ir aktuāli saistībā ar objektorientēto mantošanu – tie ir bāzes klases elementi, kas pieejami arī mantotajās klasēs, bet atšķirībā no privātajiem laukiem, tā ir tikai **nosaukumu veidošanas vienošanās**, un šādi lauki ir atklāti pieejami ar *objekts._mansaizsargatais*;
- ne slēpto, ne aizsargāto elementu nosaukumi nedrīkst beigties ar vairāk kā vienu pasvītrojuma zīmi; tas tāpēc, ka *Python* nosaukumi, kuriem abās pusēs ir pa divām pasvītrojuma zīmēm, ir ar īpašu nozīmi (piemēram, *__init__*, kā nākošajā sadaļā);

- ja runā par specifisko piekļuves nosaukumu slēptajiem elementiem *_klase__mansprivatais*, tad patiesībā tie tieši tā arī glabājas *Python* izpildes sistēmā, bet nosaukums *__mansprivatais* paliek tikai pirmkodā;
- slēpto elementu specifika attiecas arī uz klases līmeņa elementiem (sk. sadaļu 16.4.2).

16.3. Konstruktori un destruktori

Darbojoties ar objektiem objektorientētajā programmēšanā, īpaši tiek izceltas tās darbības, kas notiek sākumā (objekta inicializācija jeb sākotnējā vērtību uzstādīšana) un beigās, pirms objekts tiek likvidēts (piemēram, resursu atbrīvošana). Šīs darbības objektorientētās valodās nodrošina specifiskās metodes, sauktas attiecīgi par konstruktoru un destruktoru.

16.3.1. Konstruktors

> **Konstruktors** (*constructor*) ir specifiska klases iekšējā funkcija, kas automātiski (netieši) tiek izsaukta uzreiz pēc objekta izveidošanas.

Konstruktors tipiski tiek lietots, lai aizpildītu objektu ar sākotnējām vērtībām, to varētu saukt arī par inicializatoru.

No vienas puses, konstruktors, ja neskaita tā būtību, ir tāda pati vien metode kā citas, no otras puses, konstruktors ir nedaudz atšķirīgs no definēšanas un izmantošanas viedokļa:

- Konstruktoram ir specifisks vārds definējot – valodā *Python* tas ir *__init__* (divas pasvītrojuma zīmes pirms *init* un divas pēc),
- konstruktora izsaukšana notiek ar īpašu sintaksi (izmantojot klases vārdu),

Pirmajā piemērā (Att. 16.1) par konstruktoru **pēc būtības** kalpo metode set. Nākošajā piemērā (Att. 16.7) objekta inicializācija tiek ievietota konstruktorā.

```
class person:
    def __init__(self,name,age): # konstruktora definēšana
        self.name = name # šeit tiek izveidots lauks name
        self.age = age # šeit tiek izveidots lauks age
    def output(self): # metodes definēšana
        print(self.name,self.age)

p = person("Peter",20) # objekta izveidošana un konstruktora
                       izsaukšana
p.output()
```

```
Peter 20
```

Att. 16.7. Vienkārša klase ar konstruktoru (salīdzināt ar Att. 16.1)

16.3.2. Destruktors

> **Destruktors** (*destructor*) ir specifiska klases iekšējā funkcija, kas automātiski tiek izsaukta tieši pirms objekta likvidēšanas.

Destruktora nosaukums valodā *Python* ir *__del__* (divas pasvītrojuma zīmes pirms *del* un divas pēc).

Ja par konstruktoru ir diezgan skaidrs, kāpēc tam atrasties katrā klasē, tad destruktoram jēga ir tikai tad, ja klase izmanto kaut kādus (dinamiskus) resursus, kuri, objektam likvidējoties, ir jāatbrīvo (piemēram, jāatbrīvo izdalītā atmiņa vai jāaizver fails). Valodā *Python*, ņemot vērā tās automātisko atmiņas attīrīšanas sistēmu, destruktora pielietojums ir salīdzinoši mazāks.

Destruktors automātiski tiek izsaukts līdz ar objekta likvidēšanu, un objekta likvidēšanu var panākt šādos veidos:

- pielietojot objektam operatoru *del* (kā nākošajā piemērā Att. 16.8),
- piešķirot objekta mainīgajam citu vērtību, piemēram, *p = None*.

```
class person:
    def __init__(self,name,age): # konstruktora definēšana
        self.name = name # šeit tiek izveidots lauks name
        self.age = age # šeit tiek izveidots lauks age
    def output(self): # metodes definēšana
        print(self.name,self.age)
    def __del__(self): # destruktora definēšana
        print("Deleted:",self.name,self.age)

p = person("Peter",20) # objekta izveidošana un konstruktora
                izsaukšana
p.output()
del p # objekta likvidēšana un attiecīgi destruktora
                izsaukšana
```

```
Peter 20
Deleted: Peter 20 # destruktors
```

Att. 16.8. Vienkārša klase ar konstruktoru un destruktoru (kura loma šeit ir simboliska – tikai izdrukāt informāciju par izsaukšanas faktu)

16.4. Instances un klases elementi

16.4.1. Vispārējs apraksts

Python klasē var definēt divu veidu elementus (laukus un metodes):

- instances (objekta) līmeņa,
- klases līmeņa un statiskos.

Līdz šim aprakstītie objektu lauki un metodes bija t.s. **instances elementi** – unikāli katram objektam:

- instances lauks – pieder tieši šim objektam.
- instances metode – no tās iespējama piekļuve tieši šī objekta laukiem un citām instances metodēm.

16.4.2. Klases līmeņa elementi

Lauki un metodes var būt ne tikai piederošas konkrētam objektam, bet arī tādas, kas ir kopīgas visiem šīs klases objektiem – t.s. klases elementi jeb **klases līmeņa elementi**:

- klases lauks – ir viens uz visiem klases objektiem (pat ja neviens objekts vēl neeksistē),
- klases metode – no tās iespējama piekļuve tikai klases laukiem un citām klases metodēm, bet nevis konkrētā objekta elementiem (šādu metožu pirmais parametrs ir norāde uz klasi, nevis objektu).

Klases lauki un metodes ir līdzīgi jēdzieni attiecīgi kā **statiskie** lauki un metodes citās programmēšanas valodās, bet ar papildus tieši padotu klases kontekstu. Tai pašā laikā *Python* pastāv arī statiskās metodes, kas no klases metodēm atšķiras ar to, ka klasei pieder tikai klases strukturēšanas ietvaros, bet bez tiešas pieejas klases elementiem no metodei (ar pirmo parametru) padotās norādes.

Pie klases elementiem var piekļūt divos veidos:

- izmantojot klases norādi (klases metodes pirmais parametrs vai klases (globālais) vārds),
- izmantojot objekta norādi (instances metodes pirmais parametrs vai objekts) –
    - laukiem tas iespējams tikai lasīšanas režīmā (rakstīšanas režīmā *Python* izveidos jaunu instances lauku ar šādu pašu nosaukumu; šī atšķirība starp lasīšanas un rakstīšanas režīmu ir organizēta pēc tāda paša principa kā globālo mainīgo izmantošanā funkcijās, kā aprakstīts sadaļā 12.3.1).

Klases (līmeņa) lauku definē divos veidos:

- piešķirot vērtību klases definīcijā (ārpus jebkuras metodes, t.i., pirmajā līmenī),
- piešķirot vērtību caur klases norādi (klases metodes pirmais parametrs vai klases vārds).

Klases (līmeņa) metodi definē:

- pirms tās norādot dekoratoru *@classmethod* (dekorators ir vārds, kam priekšā ir simbols @),
- pirmais parametrs šādā metodē ir klases norāde.

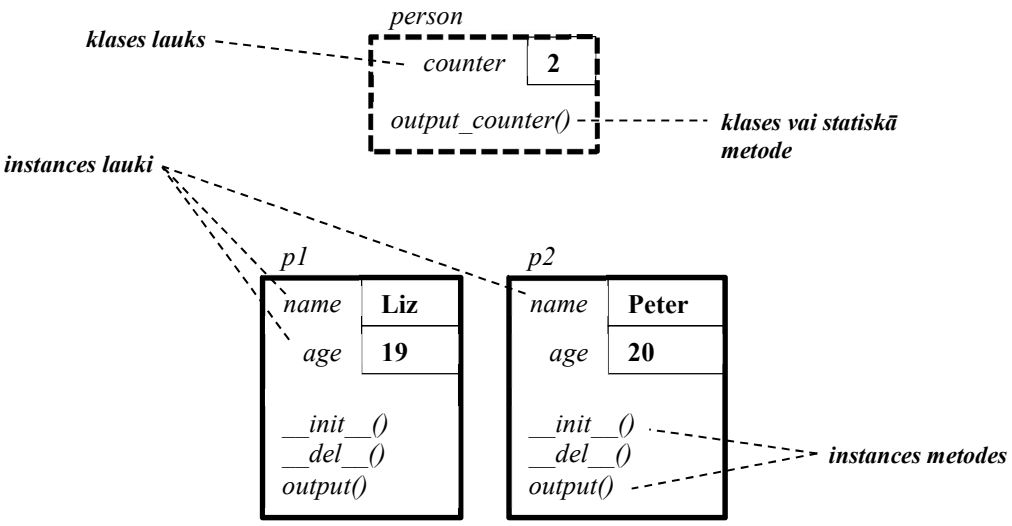
Vienkāršākais piemērs klases līmeņa elementu parādīšanai ir (vienas klases) objektu skaitītājs (Att. 16.9, vizualizācija Att. 16.10) – counter skaita, cik noteiktā momentā ir eksistējošu objektu *person*, t.i., izveidojot jaunu, pieskaita klāt, bet likvidējot – atskaita nost.

```
class person:
    counter = 0 # klases līmeņa lauks
    def __init__(self,name,age):
        self.name = name # instances lauka veidošana
        self.age = age # instances lauka veidošana
        person.counter += 1 # klases lauka modifikācija
    def __del__(self):
        person.counter -= 1 # klases lauka modifikācija
    def output(self): # instances metode
        print(self.name,self.age)
    @classmethod
    def output_counter(cls): # klases līmeņa metode
        print(cls.counter)

person.output_counter() # objektu skaits 0
p1  = person("Liz",19)
p1.output()
p1.output_counter() # objektu skaits 1
p2  = person("Peter",20)
p2.output()
p2.output_counter() # objektu skaits 2
p1 = None
p2 = None
person.output_counter() # objektu skaits 0
```

```
0
Liz 19
1
Peter 20
2
0
```

Att. 16.9. Klases līmeņa elementu piemērs – klases lauks *counter* un klases metode *output_counter* šī lauka izdrukāšanai (vizualizācija Att. 16.10)



Att. 16.10. Instances un klases lauki un metodes (atbilst kodam Att. 16.9 un Att. 16.11 – atmiņas konfigurācija pirms rindiņas "*p1 = None*"). Tehniski gan pareizi būtu tā, ka arī instances metodes pieder klasei (nevis objektam), bet konceptuāli uz to var skatīties arī šādi

16.4.3. Statiska metode kā klases metodes variācija

Statiska metode ir gandrīz tas pats, kas klases metode, tikai tai tiešā veidā netiek padots klases konteksts (kā caur pirmo parametru klases metodei), tomēr šo kontekstu var iegūt ar klases globālo nosaukumu (šeit: *person*) (pirmkods Att. 16.11). Statiskās metodes valodā *Python* savā ziņā ir nodeva jēdzienu savietojamībai ar citām valodām, kur statiskas metodes eksistē, bet valodā *Python* lielākajā daļā gadījumu labāka izvēle būs klases līmeņa metode (tā vairāk atbilst *Python* "garam"). Vēl vairāk – tas, ka *Python* vienlaicīgi ir gan statiskās, gan klases metodes, rada zināmu sajukumu, un īsti pamatotu argumentu gan klases līmeņa, gan statisku metožu vienlaicīgai eksistencei valodā nav.

```
class person:
    counter = 0 # klases līmeņa lauks
    def __init__(self,name,age):
        self.name = name # instances lauka veidošana
        self.age = age # instances lauka veidošana
        person.counter += 1 # klases lauka modifikācija
    def __del__(self):
        person.counter -= 1 # klases lauka modifikācija
    def output(self): # instances metode
        print(self.name,self.age)
    @staticmethod
    def output_counter(): # statiska metode
        print(person.counter)

person.output_counter() # objektu skaits 0
p1  = person("Liz",19)
p1.output()
p1.output_counter() # objektu skaits 1
p2  = person("Peter",20)
p2.output()
p2.output_counter() # objektu skaits 2
p1 = None
p2 = None
person.output_counter() # objektu skaits 0
```

```
0
Liz 19
1
Peter 20
2
0
```

Att. 16.11. Klases līmeņa un statisko elementu piemērs – klases lauks *counter* un statiskā metode *output_counter* šī lauka izdrukāšanai (tas pats, kas piemērā Att. 16.9, bet klases metodes vietā ir statiskā metode, kas ir iezīmēta trekināti)

Viens no statisku metožu pielietojuma piemēriem varētu būt vairāku parasto funkciju apvienojums kopējā struktūrā (tādā kā funkciju pakotnē), tādējādi ieviešot papildus nosaukumu telpas (*namespace*) līmeni (pirmkods Att. 16.12) – ja šeit izmantotu klases līmeņa metodes, būtu mākslīgi (un nevajadzīgi) jāievieš papildus parametrs.

```
class mymath:
    @staticmethod
    def add(a,b):
        return a+b
    @staticmethod
    def mult(a,b):
        return a*b

print(mymath.add(5,7))
print(mymath.mult(5,7))
```

```
12
35
```

Att. 16.12. Statiskas metodes, izmantojot klasi kā funkciju apkopojumu

Tomēr, ja šādas metodes izsauc tikai caur klasi (nevis caur izveidotu objektu), "statiskums" nav nepieciešams (sk. aprakstu sadaļā 16.1 un piemēru Att. 16.13).

```
class mymath:
    def add(a,b):
        return a+b
    def mult(a,b):
        return a*b

print(mymath.add(5,7))
print(mymath.mult(5,7))
```

```
12
35
```

Att. 16.13. Funkciju apkopojums klasē bez nosaukšanas par statiskām

Tad nu vienīgais jēdzīgais statisku metožu pielietojums ir situācijās, kad ir vajadzīgas neatkarīgas funkcijas klasēs, no kurām tiek veidoti objekti, un šīs neatkarīgās funkcijas tiek izsauktas caur šiem izveidotajiem objektiem (Att. 16.14).

```
class mymath:
    def __init__(self,a,b):
        self.a = a
        self.b = b
    def printsum(self):
        print(self.add(self.a,self.b))
    @staticmethod
    def add(a,b):
        return a+b

m = mymath(5,7)
m.printsum()
```

```
12
```

Att. 16.14. Statiskas metodes "jēdzīga" pielietojuma piemērs (pati metode ir neatkarīga no klases un klases objektiem, bet tiek izsaukta caur klases objektu)

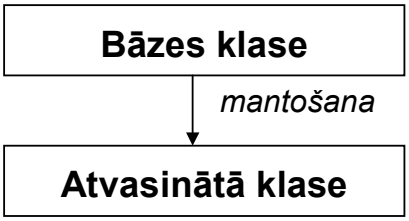
16.5. Mantošana

> **Mantošana** (*inheritance*) ir objektorientētās programmēšanas mehānisms, kad klase pārņem (manto) elementus (laukus un metodes) no jau eksistējošas klases vai pat vairākām klasēm.

Valodā Python, ņemot vērā objekta lauku veidošanas specifiku, varētu teikt, ka mantošana nozīmē tikai metožu pārņemšanu.

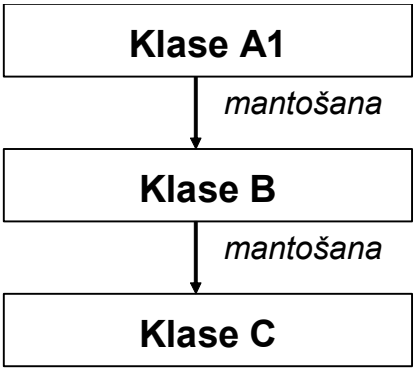
Mantošana ir vēl viens mehānisms pirmkoda vairākkārtējai izmantošanai, kas papildina koda strukturēšanas iespējas, kā arī nodrošina ērtāku uzturēšanu.

Klases, no kurām mantošanas rezultātā tiek iegūti elementi, sauc par **bāzes klasēm** (*base class*), bet klasi, kura tiek veidota, izmantojot mantošanu – par **atvasināto klasi** (*derived class*) vai **mantoto** (*inherited*) klasi.



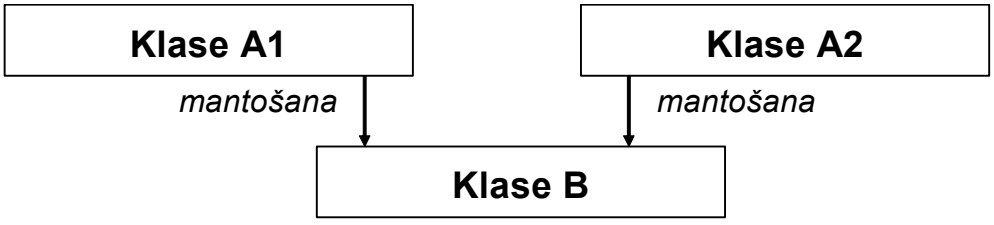
Att. 16.15. Vienkārša mantošanas shēma (koda piemēru sk. Att. 16.18)

Mantošanas process var būt kaskādes veida, līdz ar to viena un tā pati klase var būt gan atvasinātā klase, gan bāzes klase citām klasēm.



Att. 16.16. Kaskādes veida mantošanas shēma (koda piemēru sk. Att. 16.19)

Klase var mantot no vairākām citām – to sauc par daudzkāršo mantošanu (*multiple inheritance*).



Att. 16.17. Daudzkāršās mantošanas shēma (koda piemēru sk. Att. 16.19)

Klases mantošana notiek, pēc klases nosaukuma uzrādot klasi vai klases, no kurām tiks mantots (pēc līdzīgas sintakses, kā konstruktorā alternatīvajā variantā tiek inicializēti lauki).

```
class A:
    pass

class B(A): # klase B mantojas no A
    pass

a = A()
b = B()
```

Att. 16.18. Mantošanas shēma: klase B manto no klases A

```
class A1:
    pass

class A2:
    pass

class B(A1,A2): # daudzkāršā mantošana
    pass

class C(B):
    pass

a1 = A1()
a2 = A2()
b = B()
c = C()
```

Att. 16.19. Daudzkāršā mantošana (B manto no A1 un A2) un kaskādes veida mantošana (B ir gan atvasinātā, gan bāzes klase)

Nākošajā piemērā (Att. 16.20) klasē *person* ir divi lauki un attiecīgs konstruktors un izdrukas metode, bet klasē *student* ir tie paši divi lauki, kā arī vēl viens lauks klāt (*study_year*), un pirmo divu lauku uzstādīšana un izdrukāšana notiek caur bāzes klases funkcionalitāti, t.i., izmantojot mantošanas sagādātās iespējas.

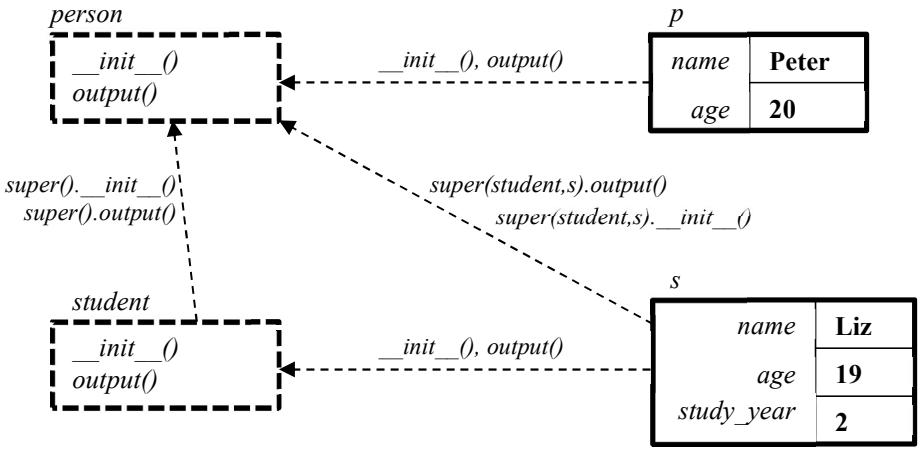
```
class person: # bāzes klase
    def __init__(self,name,age):
        self.name = name
        self.age = age
    def output(self):
        print(self.name,self.age)

class student(person): # klase student manto no klases person
    def __init__(self,name,age,study_year):
        super().__init__(name,age) # bāzes klases konstruktors
        self.study_year = study_year # šīs klases specifisks
                lauks
    def output(self): # metodes pārdefinēšana (overriding)
        super().output() # bāzes klases output
        print(self.study_year) # vēl klāt specifiskā lauka
                izdruka

p = person("Peter",20) # objekta izveidošana ar person
p.output() # personas izdruka
s = student("Liz",19,2) # objekta izveidošana ar student
s.output() # studenta izdruka
super(student,s).output() # studenta kā personas izdruka (bez
                gada) - bāzes klases output
```

```
Peter 20
Liz 19
2
Liz 19
```

Att. 16.20. Klašu mantošanas saturīgs piemērs



Att. 16.21. Metodes un pieeja tām no dažādiem objektiem un klasēm mantošanas gadījumā (atbilst kodam Att. 16.20)

Abās klasēs (Att. 16.20, Att. 16.21) ir pieejama metode ar vienādu nosaukumu *output*. Pareizās (tātad, mantotās klases) metodes izvēle starp bāzes un mantoto klasi tiek saukta par **pārdefinēšanu** vai **pārmākšanu** (*overriding*).

Kopsavilkums saistībā ar metožu **pārdefinēšanu** (*overriding*):

- ja metodi, kas ir ar vienu nosaukumu gan bāzes, gan mantotajā klasē, tiešā veidā izsauc ar mantotās klases objektu (piemēra priekšpēdējā rindiņa), tad automātiski tiek ņemta mantotās klases metode,
- ja no objekta izsauc metodi, kas ir tikai bāzes klasē, tad, protams, šī (t.i., bāzes klases) metode tad arī tiek izsaukta, jo visas bāzes klases metodes ir pieejamas no mantotās klases objektiem,
- **citās programmēšanas valodās** tas nenotiek automātiski, un lai vienmēr tiktu ņemta mantotās klases metode, attiecīgā bāzes klases metode jānosaka kā **virtuāla** (*virtual*),
- valodā *Python* tas nav nepieciešams (jo notiek automātiski), un no citu programmēšanas valodu terminoloģijas viedokļa raugoties – *Python* klasēs visas metodes ir **automātiski virtuālas**.

Ja dublējošu metožu gadījumā tomēr ir nepieciešama pieeja bāzes klases metodei, tad to var izdarīt ar speciālo metodi *super()*, kas atgriež norādi uz bāzes klasi (Att. 16.20):

- no mantotās klases iekšienes – ar *super()* bez parametriem (sk. klases *student* metodi *output*),
- no citurienes – izmantojot *super()* ar diviem parametriem – mantotā klase, objekts (sk. piemēra beidzamo rindiņu).

16.6. Citi objektorientētās programmēšanas mehānismi

16.6.1. Operatoru pārslogošana

Valodā Python ļoti plaši pielieto operatorus, un tos iespējams ieviest arī lietotāja definēto datu tipu (klašu) funkcionalitātē.

Lai to izdarītu, vispirms ir jāsaprot, ka jebkuru operatoru iespējams pierakstīt funkcionālajā pierakstā, t.i., it kā tas būtu funkcija (metode), piemēram, operatoram + atbilst metode (*__add__* vai *__radd__*).

Tādājādi

```
a - b;
```

ir pierakstāms arī šādi:

```
a.__sub__(b)
```

vai šādi:

```
b.__rsub__(a)
```

(*r* burtiņš nozīmē *reverse*, un tas nozīmē, ka pirmais arguments ir tas, kas iekavās, bet otrais, tas, kas pamata objekts, un tam ir nozīme, programmējot operatoru piesaisti funkcionalitātei).

Operatoru pārslogošanu nosaka šādi noteikumi (sk. piemērus Att. 16.22 un Att. 16.23):

- Valodā *Python* ikviena vērtība tiek glabāta kā objekts (t.sk., *int*) – viss ir objektorientēts;
- Katram operatoram atbilst iepriekš noteikta metode vai metodes, kas pieejamas dokumentācijā (bet šajā materiālā nav uzskaitītas), piemēram,
  - (+): __add__, __radd__
  - (-): __sub__, __rsub__
  - (+=): __iadd__
  - (*=): __imul__

- Pārslogojot operatoru, viens no operatora argumentiem (pēc noklusējuma – pirmais) tiek reprezentēts ar klasi (tātad, pirmo parametru (*self*) metodes realizācijā), bet otrs (ja operators ir binārs) – ar padoto parametru (tātad, otro parametru metodes realizācijā),
    - izmantojot reversās operatora metodes (nosaukums sākas ar r, salīdzinot ar parasto), operatora funkcionalitāti realizē relatīvi pret otro argumentu
- No atmiņas pārvaldības viedokļa ir divu veidu operatori,
    - tie, kas modificē eksistējošu objektu (piemēram, +=)
    - tie, kas veido jaunu objektu (piemēram, +)

```
class person:
    def __init__(self,name,age):
        self.name = name
        self.age = age
    def output(self):
        print(self.name,self.age)
    def __iadd__(self,val): # operatora += pārslogošana
        if type(val)==int:
            self.age += val # palielina gadu skaitu
        elif type(val)==str:
            self.name += " " + val # pieliek otru vārdu
        return self

p = person("Peter",20)
p.output()
p += 2
p.output()
p += "Thomas"
p.output()
```

```
Peter 20
Peter 22
Peter Thomas 22
```

Att. 16.22. Operatora += pārslogošana – tiek pamainīts pats objekts

```
class person:
    def __init__(self,name,age):
        self.name = name
        self.age = age
    def output(self):
        print(self.name,self.age)
    def __add__(self,val): # operatora + pārslogošana
        pp = person(self.name,self.age) # objekta kopija
        if type(val)==int:
            pp.age += val
        elif type(val)==str:
            pp.name += " " + val
        return pp
    def __radd__(self,val): # šeit self ir otrais arguments
        pp = person(self.name,self.age)
        if type(val)==int:
            pp.age += val
        elif type(val)==str:
            pp.name += " " + val
        return pp

p = person("Peter",20)
pp = p + 2 + "John" # šeit 2x nostrāda __add__
p.output() # vecais objekts
pp.output() # jaunais objekts ar pamainītiem abiem laukiem
ppp = 1 + pp # šeit nostrāda __radd__
ppp.output() # trešais objekts ar izmaiņām pret otru
```

```
Peter 20
Peter John 22
Peter John 23
```

Att. 16.23. Operatora + pārslogošana – tiek veidots jauns objekts, tiek izmantota arī reversā metode

16.6.2. Speciālās metodes un lauki

Valodā Python tiek plaši lietotas speciālās metodes, kas atpazīstamas pēc nosaukuma, jo tā **abās pusēs ir pa divām pasvītrojuma zīmēm**. Līdz šim apskatītās šādas metodes ir konstruktors, destruktors un operatoru pārslogošanas metodes. Šajā nodaļā doti vēl daži izvēlēti piemēri speciālo elementu izmantošanai – gan gatavie jau iebūvētie gan pārdefinējamie.

Objekta (arī klases) elementus var apskatīt (kā arī izmainīt to vērtības), izmantojot iebūvēto speciālo lauku *__dict__* (Att. 16.24). Pēc šī piemēra var redzēt principu, kā uzbūvēti objekti Python atmiņā.

```
class person:
    def __init__(self,name,age):
        self.name = name
        self.age = age
    def output(self):
        print(self.name,self.age)

p = person("Peter",20)
print(p.__dict__) # lauki ar vērtībām
p.__dict__["name"] = "John" # vērtības nomaiņa
print(p.__dict__) # lauki ar vērtībām
p.output()
```

```
{'name': 'Peter', 'age': 20}
{'name': 'John', 'age': 20}
John 20
```

Att. 16.24. Speciālā lauka *__dict__* izmantošana piekļuvei objekta elementiem

Ja klasē ir definēta metode *__len__*, tad drīkst izmantot *len(objekts)* tās izsaukšanai, bet, ja metode *__call__*, tad pats objekts ar iekavām *objekts()* tās izsaukšanai, savukārt metode __str__ pārveido objektu uz tekstu (izmantojot *str(objekts)*), t.sk., automātiski tiek izsaukta, izmantojot *print* (Att. 16.25).

```
class person:
    def __init__(self,name,age):
        self.name = name
        self.age = age
    def __str__(self): # teksta reprezentācija
        return self.name+', '+str(self.age)
    def __call__(self): # drukāšanas metode kā objekts()
        print(self.name,self.age)
    def __len__(self): # vecuma atgriešana kā len()
        return self.age

p = person("Peter",20)
print(p) # __str__
p() # __call__
print(len(p)) # __len__
```

```
Peter, 20
Peter 20
20
```

Att. 16.25. Speciālo metožu *__len__*, *__str__* un *__call__* izmantošana

16.6.3. Iterējami objekti

Objekts ir **iterējams** (*iterable*), ja tā elementiem var pārstaigāt (**iterēt**) veidā:

```
for elem in objekts
```

Ja mēģina iterēt neiterējamu objektu, tad parādās kļūda:

```
builtins.TypeError: 'person' object is not iterable
```

Lai padarītu objektu iterējamu, klasē jānodefinē divas papildus metodes:

- *__iter__*, kas pasaka, ka objekts principā ir iterējams un potenciāli uzstāda iteratora sākuma pozīciju (ja objektu paredzēts iterēt tikai vienu reizi, tad sākuma pozīciju var uzstādīt arī konstruktorā *__init__*),
- *__next__*, kas
  - atgriež kārtējo elementu,
  - pārbīda iekšējo norādi uz nākošo pozīciju,
  - sasniedzot beigas, signalizē (ar *raiseStopIteration*) – ja šīs sadaļas nav, tad iterējamu objektu nedrīkst darbināt ar *for* ciklu, kas tad novestu pie ieciklošanās, bet tikai tiešā veidā ar *__next__*, apstāšanos nodrošinot ārēji.

Nākošajā piemērā iterators programmēts uz personas vārda, atgriežot to pa vienam burtam (Att. 16.26).

```
class person:
    def __init__(self,name,age):
        self.name = name
        self.age = age
    def __iter__(self):
        self.iterpos = 0 # iteratora pozīcija
        return self
    def __next__(self):
        if self.iterpos>=len(self.name):
            raise StopIteration # sasniegtas beigas
        else:
            elem = self.name[self.iterpos] # kārtējais
                 elements
            self.iterpos += 1 # nākošā pozīcija
            return elem

p = person("Peter",20)
for c in p: # iterēšana pa vārda burtiem
    print(c)
```

```
P
e
t
e
r
```

Att. 16.26. Iterējamas struktūras izveidošana ar *__iter__* un *__next__*

Iterējami objekti daļēji pārklāj funkcionalitāti, kuru nodrošina funkcijas-ģeneratori (sk. sadaļu 15.3), tomēr šie abi mehānismi var viens otru papildināt, ja objektu nepieciešams iterēt vairāk nekā vienā veidā (Att. 16.27).

```
class person:
    def __init__(self,name,age):
        self.name = name
        self.age = age
    def __iter__(self):
        self.iterpos = 0 # iteratora pozīcija
        return self
    def __next__(self):
        if self.iterpos>=len(self.name):
            raise StopIteration # sasniegtas beigas
        else:
            elem = self.name[self.iterpos] # kārtējais
                 elements
            self.iterpos += 1 # nākošā pozīcija
            return elem
    def items(self): # ģenerators
        i = 0 # iteratora pozīcija
        while i<len(self.name):
            elem = self.name[i] # kārtējais elements
            yield i,elem
            i += 1 # nākošā pozīcija

p = person("Peter",20)
for c in p: # iterēšana pa vārda burtiem
    print(c,end=' ')
print()
for ic in p.items(): # iterēšana pa pozīcijām un burtiem
    print(ic)
for c in p: # iterēšana pa vārda burtiem
    print(c,end=' ')
print()
```

```
P e t e r
(0, 'P')
(1, 'e')
(2, 't')
(3, 'e')
(4, 'r')
P e t e r
```

Att. 16.27. Iterējama struktūra ar papildus metodi-ģeneratoru (*items*)

17. Darbs ar failiem

17.1. Failu apstrādes principi

> **Fails** (*file*) ir datu kopums, kas izvietots sekundārajā atmiņā un kas operētājsistēmas līmenī tiek identificēts ar noteiktu faila vārdu.

Tāpat kā operatīvajai atmiņai, arī failam **mazākā adresējamā vienība ir 1 baits**, tādējādi no datu apstrādes viedokļa var uzskatīt, ka fails ir baitu virkne.

Kaut arī darbs ar failiem valodā Python ir standartizēts, tomēr ir atsevišķas nianses, kas nosaka atšķirības failu apstrādē dažādās operētājsistēmās, jo no programmas viedokļa fails ir operētājsistēmas pakalpojums.

Failu Python programmā identificē **faila objekts**, kuru reprezentē **faila mainīgais**, kurš var būt viens no tipiem:

Galvenās darbības ar failu ir:

- lasīšana (*reading*),
- rakstīšana (*writing*).

Ņemot vērā to, ka fails ir operētājsistēmas pakalpojums, turklāt failu sistēmas līmenī tas tiek identificēts ar faila vārdu, turpretī programmā – ar faila mainīgo, faila apstrādē ir nepieciešamas divas šādas papildus darbības:

- **Faila atvēršana** (*opening*) pirms darba sākšanas (fiziskā faila piesaiste faila mainīgajam un noteikta darba režīma uzstādīšana).
- **Faila aizvēršana** (*closing*), darbu beidzot.

Faila atvēršana un aizvēršana ir saistīta ar noteiktu, pietiekoši apjomīgu darbību veikšanu operētājsistēmas līmenī, jo **fails ir resurss, par kuru atbild operētājsistēma** (piemēram, lai divi lietotāji reizē nevarētu rakstīt vienā failā).

Atverot failu, tā vārdu var uzrādīt:

- absolūti (piemēram, "C:/src/files.py"),
- relatīvi (pret kārtējo direktoriju, kas parasti ir programmas palaišanas direktorija), piemēram, "out.txt", "outdir/out.txt", "../data/maindata/out.txt"

Speciālie simboli failu nosaukumos:

- ″/″ – direktoriju atdalītājs (der arī *Windows* sistēmā),
- ″¥¥″ – direktoriju atdalītājs *Windows* sistēmā, nozīmē vienu atpakaļsvītru,
- r″¥″ – direktoriju atdalītājs *Windows* sistēmā,
- ″..″ – vienu līmeni uz augšu direktoriju struktūrā.

Faila atvēršana notiek, izmantojot metodi *open()*:

```
f = open("test.txt")
```

Pēc noklusēšanas fails tiek atvērts lasīšanas režīmā, tomēr var norādīt otru parametru, lai tieši uzrādītu režīmu.

```
f = open("test.txt","w")
```

Python faila atvēršanas režīms tiek uzrādīts kā simbolu virkne.

Tab. 17.1.

Faila atvēršanas režīmi (jāpielieto kā simbolu virknes)

Režīms/režīmi	Apraksts
`r`	(Noklusētais režīms) atver failu lasīšanas režīmā, faila norāde tiek nolikta faila sākumā.
`w`	Atver failu rakstīšanas režīmā; eksistējošu failu pārraksta; neeksistējošu izveido no jauna.
`a`	Atver failu papildināšanai beigās; neeksistējošu izveido no jauna.
`rb wb ab`	Tas pats, kas attiecīgi r, w, a, bet binārā formātā.
`r+ w+ a+`	atver failu gan lasīšanai, gan rakstīšanai, par pamatu ņemot attiecīgi režīmus r, w, a.
`rb+ wb+ ab+`	Tas pats, kas attiecīgi r+, w+, a+, bet binārā formātā.

Pēc darba ar failu tas noteikti jāaizver (ar to cita starpā tiek ziņots operētājsistēmai, ka programmai fails vairs nav vajadzīgs), ko dara metode *close()*.

```
f.close()
```

Teksta faila atvēršana *UTF-8* kodējumā notiek, izmantojot papildus parametru *encoding*:

```
f =open("test.txt", encoding="utf-8")
f2 =open("test.txt", "w", encoding="utf-8")
```

17.2. Teksta failu apstrāde

> **Teksta fails** (*text file*) ir fails, kas sastāv no simboliem (burtiem, cipariem, pieturzīmēm utt.).

Teksta fails ir atpazīstams pēc tā, ka tajā atrodamā informācija cilvēkam ir viegli uztverama, apskatot to ar vienkāršu teksta redaktoru (piemēram, *notepad*).

No programmēšanas viedokļa drīzāk runā nevis par teksta failu, bet gan par faila apstrādi teksta režīmā.

Faila apstrāde teksta režīmā parasti notiek **pa rindiņai**. Dalīšana rindās ir teksta strukturēšanas veids, kas atbilst tam, kā to dara cilvēks, pierakstot dabisko valodu tekstus.

Faila **nolasīšana** notiek,

- iterējot pa vienai rindai pa faila objektu (Att. 17.1, Att. 17.2) vai
- nolasot visu failu sarakstā ar funkciju *readlines()* (Att. 17.3).

```
f = open("in.txt","r")
for s in f: # s: kārtējā rindiņa
    print(s,end='')
f.close()
```

in.txt: (ievade) (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)

```
This¶
is·an¶
example.§
```

konsole: (izvade)

```
This
is an
example.
```

Att. 17.1. Teksta faila nolasīšana pa rindai un izvade uz ekrāna (tā kā rinda tiek nolasīta, ieskaitot ‘\n’, tad, izdrukājot rindu, to neliek)

```
for s in open("in.txt","r"): # fails nav jāaizver
    print(s,end='')
```

in.txt: (ievade) (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)

```
This¶
is·an¶
example.§
```

konsole: (izvade)

```
This
is an
example.
```

Att. 17.2. Teksta faila nolasīšana pa rindai īsajā variantā, tieši neveidojot faila objektu

```
f = open("in.txt","r")
ss = f.readlines() # saraksta no rindiņām izveide
print(ss)
f.close()
```

in.txt: (ievade) (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)

```
This¶
is·an¶
example.§
```

konsole: (izvade)

```
['This\n', 'is an\n', 'example.']
```

Att. 17.3. Visa faila nolasīšana sarakstā ar vienu komandu

Teksta **ierakstīšana** failā notiek

- tāpat kā uz konsoli, ar funkciju *write()* (sk. sadaļu 5.1),
- bet ar **papildus parametru** *file*, ar ko norāda faila objektu (Att. 17.4).

```
f = open("out.txt","w")
print("This is",file=f)
f.close()
```

out.txt: (izvade) (‘·’ tukšums, ‘¶’ jaunas rindiņas simbols, ‘§’ faila beigas)

```
This·is¶
my·great¶
example.¶§
```

Att. 17.4. Rakstīšana teksta failā ar *print*

17.3. Python datu efektīva saglabāšana failā

Šajā sadaļā tiks aprakstīta *Python* objektu ērta saglabāšana failā, automātiski veicot to serializāciju.

> **Serializācija** (*serialization*) ir (operatīvajā atmiņā esošo) datu pārveidošana saglabājamā formātā (piemēram, ierakstīšanai failā).

Tiks apskatītas divas bibliotēkas datu saglabāšanai:

- *pickle* – objektu saglabāšanai bināri,
- *json* – objektu saglabāšana teksta formātā.

17.3.1. Bibliotēka *pickle*

Bibliotēka *pickle* ļauj saglabāt vienu vai vairākus objektus pēc kārtas failā binārā formā.

Lai saglabātu objektus failā ar *pickle* (Att. 17.5):

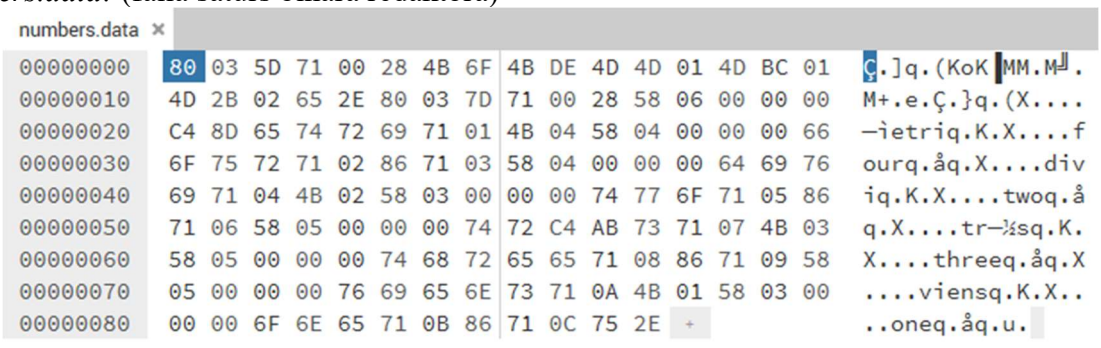
- jāatver fails binārā rakstīšanas režīmā (režīms *“wb”*),
- katra objekta saglabāšanai pēc kārtas lieto metodi *dump()*,
- beigās obligāti aizvērt failu ar *close()*.

```
import pickle

aa = [111,222,333,444,555]
dd = {'četri':(4,'four'), 'divi':(2,'two'),
                  'trīs':(3,'three'), 'viens':(1,'one')}

f = open("numbers.data","wb")
pickle.dump(aa,f) # pirmā objekta saglabāšaana
pickle.dump(dd,f) # otrā objekta saglabāšana
f.close()
```

numbers.data: (faila saturs binārā redaktorā)



Att. 17.5. Objektu saglabāšana ar *pickle*

Lai ielādētu saglabātos objektus no faila ar *pickle*, rīkojas simetriski saglabāšanai (Att. 17.6):

- jāatver fails binārā lasīšanas režīmā (režīms *"rb"*),
- katra objekta ielādēšanai pēc kārtas lieto metodi *load()* – objekti tiek ielādēti pa vienam tādā secībā, kā tika saglabāti,

```
import pickle

f = open("numbers.data","rb")
x = pickle.load(f) # pirmā objekta ielādēšana
y = pickle.load(f) # otrā objekta ieladēšana
f.close()

print(x)
print(y)
```

```
[111, 222, 333, 444, 555]
{'četri': (4, 'four'), 'divi': (2, 'two'), 'trīs': (3,
                  'three'), 'viens': (1, 'one')}
```

Att. 17.6. Objektu ielādēšana no faila ar *pickle*

17.3.2. Bibliotēka *json*

Bibliotēka *json* ļauj saglabāt objektu failā teksta formā pēc līdzīgas sintakses kā ar *pickle*.

Lai saglabātu objektu failā ar *json* (Att. 17.7):

- jāatver fails rakstīšanas režīmā (režīms *"w"*),
- objekta saglabāšanai lieto metodi *dump()* (ja objekti ir vairāki, no tiem var izveidot vienu struktūru),
- beigās obligāti aizvērt failu ar *close()*.

```
import json

aa = [111,222,333,444,555]
dd = {'četri':(4,'four'), 'divi':(2,'two'),
                  'trīs':(3,'three'), 'viens':(1,'one')}

f = open("numbers.txt","w")
json.dump((aa,dd),f) # abu objektu saglabāšana kopējā
                  struktūrā
f.close()
```

numbers.txt:

```
[[111, 222, 333, 444, 555], {"\u010detri": [4, "four"],
 "divi": [2, "two"], "tr\u012bs": [3, "three"], "viens": [1,
 "one"]}]
```

Att. 17.7. Objektu saglabāšana ar *json*

Lai ielādētu saglabātos objektus no faila ar *json*, rīkojas simetriski saglabāšanai (Att. 17.8):

- jāatver fails lasīšanas režīmā (režīms *"r"*),
- objekta ielādēšanai lieto metodi *load()*.

```
import json

f = open("numbers.txt","r")
x,y = json.load(f) # abu objektu ielādēšana
f.close()

print(x)
print(y)
[111, 222, 333, 444, 555]
{'četri': (4, 'four'), 'divi': (2, 'two'), 'trīs': (3,
                  'three'), 'viens': (1, 'one')}
```

Att. 17.8. Objektu ielādēšana no faila ar *json*

18. *Python* programmas failu struktūra – moduļi un pakotnes

18.1. Funkcijas darbam ar failiem

Bibliotēkā *os* ir pieejamas vairākas funkcijas darbam ar failiem un direktorijām:

- ***getcwd*** – (*get current working directory*) atgriež kārtējo direktoriju,
- ***chdir*** – (*change directory*) nomaina direktoriju,
- ***mkdir*** – (*make directory*) izveido direktoriju,
- ***rmdir*** – (*remove directory*) izdzēš tukšu direktoriju,
- ***listdir*** – (*list directory*) atgriež direktorijas objektu vārdus,
- ***path.exists*** – nosaka, vai objekts (fails vai direktorija) eksistē,
- ***path.isdir*** – nosaka, vai objekts ir direktorija (nevis fails),
- ***path.isfile*** – nosaka, vai objekts ir fails (nevis direktorija),
- ***path.abspath*** – no relatīvā ceļa izrēķina absolūto.

Faila vai direktorijas vārdu var uzrādīt:

- absolūti (piemēram, *"C:/src/myproj/main/files.py"*),
- relatīvi (pret kārtējo direktoriju, kas parasti ir programmas palaišanas direktorija), piemēram, *"out.txt"*, *"outdir/out.txt"*, *"../../data/maindata/out.txt"*

Speciālie simboli failu nosaukumos:

- *"/"* – direktoriju atdalītājs (der arī *Windows* sistēmā),
- *"\\"* – direktoriju atdalītājs *Windows* sistēmā, nozīmē vienu atpakaļsvītru,
- *r"\"* – direktoriju atdalītājs *Windows* sistēmā,
- *".."* – vienu līmeni uz augšu direktoriju struktūrā.

```
C:/
└───src/
    ├───aux/
    ├───myproj/
    │   ├───main/
    │   │   ├───data/
    │   │   │   ├───file1.txt
    │   │   │   ├───file2.txt
    │   │   │   └───file3.txt
    │   │   ├───filetools.py
    │   │   └───importtest.py
    │   └───utils/
    └───testdir/
```

Att. 18.1. Direktoriju un failu struktūra pirms piemēra (Att. 18.2) demonstrēšanai

```
import os

print(os.getcwd()) # kārtējā direktorija
print(os.path.exists('NODATA/NOFILE.TXT'),os.path.exists('data
                /file1.txt'))
print(os.path.isdir('data'),os.path.isdir('filetools.py'))
print(os.path.isfile('data'),os.path.isfile('filetools.py'))
print(os.path.abspath("../.")) # izdrukā C:\src
os.chdir("../..") # aiziet uz C:\src
print(os.getcwd()) # kārtējā direktorija: C:\src
print("'C:/src' contents", os.listdir())
print("'C:/src/myproj/main/data' contents:")
print(os.listdir('myproj/main/data'))
os.rmdir('testdir')
print("'C:/src' contents", os.listdir())
os.mkdir('TESTDIR2')
print("'C:/src' contents", os.listdir())
```

```
C:\src\myproj\main
False True
True False
False True
C:\src
C:\src
'C:/src' contents ['aux', 'myproj', 'testdir']
'C:/src/myproj/main/data' contents:
['file1.txt', 'file2.txt', 'file3.txt']
'C:/src' contents ['aux', 'myproj']
'C:/src' contents ['aux', 'myproj', 'TESTDIR2']
```

Att. 18.2. Funkcijas darbam ar failiem (atrodas failā *filetools.py* atbilstoši Att. 18.1)

```
C:/
└──src/
   ├──aux/
   ├──myproj/
   │  ├──main/
   │  │  ├──data/
   │  │  │  ├──file1.txt
   │  │  │  ├──file2.txt
   │  │  │  └──file3.txt
   │  │  ├──filetools.py
   │  │  └──importtest.py
   │  └──utils/
   └──TESTDIR2/
```

Att. 18.3. Direktoriju un failu struktūra pēc piemēra (Att. 18.2) darbināšanas

18.2. Moduļi un to importēšana

18.2.1. Galvenie principi un standarta moduļu imports

Par **moduli** valodā *Python* sauc jebkuru failu, kurā atrodas programma, piemēram, *importtest.py* (kā piemērā Att. 18.3), t.sk. jebkuru programmas failu standartbibliotēkā.

Lai citu moduļu objektus (klases, funkcijas, mainīgos) izmantotu aktuālajā modulī, jāveic to imports.

Citu moduļu importēšanu izmantošanai programmā var veikt divos veidos (Att. 18.4):

- ar komandu *import*:
  - *import* ***mylibrary*** – objektiem piekļūst ar ***mylibrary****.object*
  - *import* ***mylibrary*** *as* ***alias*** – objektiem piekļūst ar ***alias****.object*
- ar komandu *from ... import*:
  - *from* ***mylibrary*** *import* ***obj1****,* ***obj2*** – objektiem piekļūst ar ***obj1****,* ***obj2***
  - *from* ***mylibrary*** *import* ******* – objektiem piekļūst ar *obj1, obj2*
  - *from* ***mylibrary*** *import* ***obj1*** *as* ***alias1****,* ***obj1*** *as* ***alias1*** – objektiem piekļūst ar ***alias1****,* ***alias2***

```
import math
print(math.pi)

import math as m
print(m.pi)

from math import pi,pow
print(pi,pow(pi,2)) # PI kvadrātā

from os import *
print(getcwd()) # kārtējā direktorija

from os import getcwd as wd
print(wd())
```

```
3.141592653589793
3.141592653589793
3.141592653589793 9.869604401089358
C:\src\myproj\main
C:\src\myproj\main
```

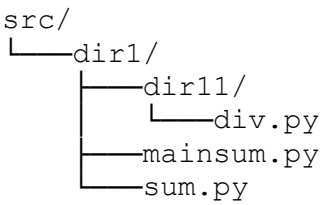
Att. 18.4. Standarta bibliotēkas moduļu (*math*, *os*) importēšana (kods failā *importtest.py* atbilstoši Att. 18.3)

18.2.2. Absolūtā importēšana

Importēšanu var veikt arī starp saviem veidotajiem moduļiem.

Absolūtā importēšana nodrošina moduļu importēšanu šādās lokācijās:

- no kārtējās direktorijas (piemēram, ***mymodule***) vai
- dziļākas direktorijas (piemēram, ***dirx.diry.mymodule***) – par direktoriju atdalītāju tiek lietots punkts (.).



Att. 18.5. Absolūtā importēšana – failu struktūra ar galveno failu *mainsum.py* (sk. kodu Att. 18.6)

mainsum.py:

```
from sum import summa # no tās pašas direktorijas
print(summa(13,5))

from dir11.div import divide # vienu līmeni dziļāk
print(divide(13,5))
```

sum.py:

```
def summa(a,b):
    return a+b
```

dir11/div.py:

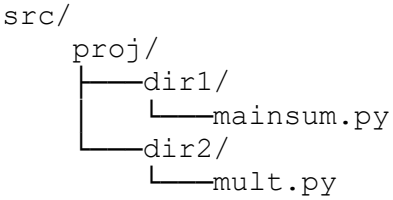
```
def divide(a,b):
    return a/b
```

```
18
2.6
```

Att. 18.6. Absolūtā importēšana no moduļa *mainsum.py* (sk. struktūru Att. 18.5)

18.2.3. Importēšana caur vecāku (*parent*) direktoriju

Šajā sadaļā aprakstīts viens no variantiem, kā importēt moduli no blakus direktorijas (sk. struktūru Att. 18.7).



Att. 18.7. Importēšana no blakus direktorijas (sk. kodu Att. 18.8)

Tiešā veidā tas nav iespējams, jo *Python* nespēj atrast moduli *mult* (Att. 18.8).

dir1/mainsum.py:

```
from mult import multi
print(multi(7,5))
```

dir2/mult.py:

```
def multi(a,b):
    return a*b
```

```
Traceback (most recent call last):
  File "c:/src/proj/dir1/mainsum.py", line 1, in <module>
    from mult import multi
builtins.ModuleNotFoundError: No module named 'mult'
```

Att. 18.8. Importēšana no blakus direktorijas (sk. struktūru Att. 18.7) – tiešā veidā nav iespējama

Lai varētu importēt moduli no blakus direktorijas, nepieciešams to (blakus direktoriju) ievietot sistēmas ceļu sarakstā *sys.path* (Att. 18.9), izdarot tā, ka *Python* par to uzzina.

dir1/mainsum.py:

```
import sys,os
multdir = os.path.abspath('../dir2') # iegūst direktoriju
sys.path.append(multdir) # piereģistrē to sys.path
from mult import multi # tagad multi atrodas "zināmā" ceļā
print(multi(7,5))
```

dir2/mult.py:

```
def multi(a,b):
    return a*b
```

```
35
```

Att. 18.9. Importēšana no blakus direktorijas (sk. struktūru Att. 18.7), vispirms aizpildot *sys.path*

Tādējādi var papildināt, ka absolūtā importēšana nodrošina moduļu importēšanu:

- no kārtējās direktorijas vai jebkuras direktoriju sarakstā ***sys.path* reģistrētas direktorijas**, kā arī
- attiecīgi pret kādu no šīm direktorijām dziļākas direktorijas.

18.2.4. Programmas galvenais fails un objekts __name__

Programmas (un valodā *Python* programma ir vienkārši failu kopums) galvenais fails ir tas, no kura tiek sākta programmas izpilde.

```
src/
    nametest/
    ├──main.py
    └──sum.py
```

Att. 18.10. Programmas struktūra galvenā faila testēšanai

Objekts *__name__* ir tas, ar kura palīdzību var noskaidrot, vai dotajā izpildē attiecīgais fails ir galvenais vai nē (Att. 18.11):

- ja galvenais – atgriež *‘__main__’*,
- citādi – faila (kā moduļa, bez *‘.py’*) vārdu.

main.py:

```
print("This file:",__name__)
from sum import summa
print(summa(8,9))
```

sum.py:

```
print("This file:",__name__)
def summa(a,b):
    return a+b
```

a) palaists main.py:

```
This file: __main__
This file: sum
17
```

b) palaists sum.py:

```
This file: __main__
```

Att. 18.11. Moduļa vārds programmas izpildē un galvenais fails (sk. struktūru Att. 18.10)

Šo īpašību var izmantot, piemēram, ja modulim jāuzvedas atšķirīgi atkarībā no tā, vai tas ir galvenais – tipisks variants, ka tiek izdalīts specifisks kods, kurš izpildās tikai, ja fails (modulis) ir galvenais (Att. 18.12).

main.py:

```
print("This file:",__name__)
from sum import summa
def main():
    print(summa(8,9))
if __name__ == '__main__':
    print('MAIN PROGRAM')
    main()
```

sum.py:

```
print("This file:",__name__)
def summa(a,b):
    return a+b
if __name__ == '__main__':
    print("TEST SUMMA(5,7)",summa(5,7))
```

a) palaists main.py:

```
This file: __main__
This file: sum
MAIN PROGRAM
17
```

b) palaists sum.py:

```
This file: __main__
TEST SUMMA(5,7) 12
```

c) palaists no Python konsoles, pieņemot, ka kārtējā direktorija ir C:/src/nametest:

```
>>> import os
>>> os.getcwd()
    'C:\\src\\nametest'
>>> import main
    This file: main
    This file: sum
>>> main.main()
    17
```

Att. 18.12. Moduļa atšķirīga uzvedība atkarībā no būšanas vai nebūšanas galvenā statusā (sk. struktūru Att. 18.10) – (a) netiek darbināts *sum.py* moduļa '__main__' bloks, (b) tiek darbināts *sum.py* moduļa '__main__' bloks, (c) ar komandu "import main" netiek darbināts ne *sum.py* moduļa '__main__' bloks, ne *main.py* moduļa '__main__'

18.3. Pakotnes un relatīvā importēšana

Pakotne ir jebkurš direktoriju koks ar *Python* kodu.

(*Python2* bija obligāti, ka katrā pakotnes direktorijā ir fails *__init__.py*, bet *Python3* tas vairs nav obligāti.)

18.3.1. Relatīvā importēšana pakotnes ietvaros

Cits variants, kā importēt moduli ne tikai "no apakšas" – ir uztvert noteiktu direktorijas koku kā **pakotni** (*package*), un tad šīs pakotnes ietvaros ar speciāli sintaksi iespējama importēšana arī caur vecāka direktoriju, respektīvi, t.s. **relatīvā importēšana**. Šim nolūkam:

- jāsaprot, kurš ir **direktoriju koks, kas būs pakotne** – šeit *proj* (Att. 18.13),
- sākotnējai piekļuvei pakotnei jābūt "no ārpuses", izmantojot absolūto importēšanu (šeit: fails *main.py* un *main2.py*, kas ir ārpus *proj* direktorijas,

- visās *import* komandās attiecībā uz šīs pakotnes moduļiem sākumā jāliek viens vai vairāki **punkti** (viens punkts – kārtējā direktorija, katrs nākamais punkts – viens līmenis uz augšu) – sk. failu *badmain.py* un *mainsum.py* saturu kodā Att. 18.14,
- punkti ir tie, kas tīri **vizuāli** atšķir relatīvo importēšanu no absolūtās.

```
src/
├──main.py
├──main2.py
└──proj/
    ├──badmain.py
    ├──dir1/
    │   └──mainsum.py
    └──dir2/
        └──mult.py
```

Att. 18.13. Pakotne *proj* relatīvās importēšanas demonstrēšanai

main.py:
```
from proj.dir1.mainsum import myfun # absolūtā importēšana
myfun()
```
main2.py:
```
import proj.badmain # absolūtā importēšana
```
proj/badmain.py:
```
from .dir1.mainsum import myfun # relatīvā importēšana
myfun()
```
proj/dir1/mainsum.py:
```
from ..dir2.mult import multi # relatīvā importēšana
def myfun():
    print(multi(7,5))
```
proj/dir2/mult.py:
```
def multi(a,b):
    return a*b
```
d) palaists badmain.py:
```
Traceback (most recent call last):
  File "c:/src/proj/badmain.py", line 1, in <module>
    from .dir1.mainsum import myfun
builtins.ImportError: attempted relative import with no known
                      parent package
```
e) palaists main.py:
```
35
```
f) palaists main2.py:
```
35
```

Att. 18.14. a) Neveiksmīga relatīvā importēšana, palaižot *badmain.py*, b) veiksmīga relatīvā importēšana, palaižot *main.py*, b) veiksmīga relatīvā importēšana, palaižot *main2.py*

Ievērojiet, ka iepriekšējā piemērā (Att. 18.14), palaižot programmu:

- par pakotni tiek uzskatīta direktorija *proj*,
- programmas palaišana notiek ārpus (nevis iekš) pakotnes (*main.py* vai *main2.py*)
- palaižot *badmain.py* (gadījums (a)), sanāk, ka tiek palaists no pakotnes *proj* "iekšpuses", tāpēc ir kļūda – no pakotnes iekšpuses var importēt, nevis palaist.
- palaižot *main.py* (gadījums (b)), programma veiksmīgi nostrādā, izveidojot importēšanas ķēdīti: *main→mainsum(myfun)→mult(multi)*,

- palaižot *main2.py* (gadījums (c)), programma arī veiksmīgi nostrādā, izveidojot importēšanas ķēdīti: *main→badmain→mainsum(myfun)→mult(multi)*, tātad pašam *badmain.py* nav ne vainas, tikai viņu nedrīkst tieši izsaukt no ārpuses, kas nozīmētu relatīvo importēšanu no ārpuses (no *badmain*),
- tātad relatīvā importēšana iespējama tikai pakotnes iekšienē, nevis no ārpuses, un moduli, kurā ir relatīvā importēšana, nedrīkst tiešā veidā izsaukt.

18.3.2. Pakotnes un funkcija *__init__.py*

Kaut arī *Python3* vairs nav obligāti, ka pakotnes direktorijās atrodas moduļi *__init__.py*, tomēr tos tur var ievietot, un tie **automātiski izpildās**, pirmo reizi ielādējot **kādu moduli** attiecīgajā direktorijā. Piemērā papildus izmantosim iebūvēto objektu *__file__*, kas atgriež attiecīgā programmas faila vārdu.

```
src/
├──main.py
└──proj/
    ├──__init__.py
    ├──dir1/
    │   ├──__init__.py
    │   └──mainsum.py
    └──dir2/
        ├──__init__.py
        └──mult.py
```

Att. 18.15. Pakotne *proj* ar moduļiem *__init__.py* (kods Att. 18.16)

main.py:
```
from proj.dir1.mainsum import myfun # absolūtā importēšana
myfun()
```
proj/dir1/mainsum.py:
```
from ..dir2.mult import multi # relatīvā importēšana
def myfun():
    print(multi(7,5))
```
proj/dir2/mult.py:
```
def multi(a,b):
    return a*b
```
__init__.py *(visu 3 failu saturs vienāds):*
```
print(__file__)
```
palaists main.py:
```
c:\src\proj\__init__.py
c:\src\proj\dir1\__init__.py
c:\src\proj\dir2\__init__.py
35
```

Att. 18.16. Moduļu *__init__.py* automātiska izsaukšana pakotnē (Att. 18.15)

Tab. 18.1.

Importēšanas īsais kopsavilkums ar piemēru (ja importē failu *mymodule.py*)

Importējamā faila relatīvā atrašanās	**Piemērs**	**Apraksts**
Absolūtā importēšana		
• kārtējā faila direktorija • sarakstā *sys.path* reģistrēta direktorija	`mymodule`	kādā no attiecīgajām direktorijām
	`dir1.mymodule`	vienu līmeni dziļāk
	`dir1.dir2.mymodule`	divus līmeņus dziļāk
Relatīvā importēšana (tikai pakotnes ietvaros) – ar punktu priekšā		
• kārtējā faila direktorija	`.mymodule`	šajā direktorijā
	`.dir1.mymodule`	vienu līmeni dziļāk
	`.dir1.dir2.mymodule`	divus līmeņus dziļāk
	`..mymodule`	vienu līmeni augstāk
	`...dir1.dir2.mymodule`	divus līmeņus augstāk, tad divus dziļāk

19. Izņēmumsituāciju apstrāde

19.1. Definīcija un piemēri

> **Izņēmumsituācija** (*exception*) ir ārpus normālās plūsmas loģikas esošs programmas izpildes notikums, kas apstrādājams ar speciālu izņēmumsituāciju apstrādes mehānismu (*exception handling*).

Izņēmumsituācija (jeb izņēmums) ir kaut kas "pa vidu" starp kļūdu un normālu programmas darbību.

Piemēram, nepareizas paroles ievadīšana –

- nav sistēmas kļūda (bet gan lietotāja kļūda) – tas ir pat visai parasts notikums,
- tai pat laikā tā nav arī normāla darbība, jo mērķis, sākot strādāt ar kādu sistēmu, tomēr ir tai veiksmīgi pieslēgties.

Tomēr tā savā ziņā ir gan kļūdaina situācija, gan normāla darbība, un ir gadījumi, kad nav viennozīmīgi skaidrs, vai dotais notikums būtu uzskatāms par izņēmumsituāciju vai specifisku, bet tomēr normālu situāciju, piemēram, persona grib sistēmā izveidot kontu, bet personas piedāvātais pieslēgšanās vārds jau ir izmantots citam eksistējošam kontam sistēmā.

Modernajās programmēšanas valodās, t.sk. *Python*, šādu specifisku situāciju apstrādei tiek ieviests īpašs **izņēmumsituāciju apstrādes mehānisms**, kas

- nodrošina specifisku (parasti retāk notiekošu) situāciju **citādāku** apstrādi,
- ietekmē programmas struktūru, izņēmumsituāciju apstrādi **nodalot** no pārējās programmas.

Tas, kas izņēmumsituāciju apstrādes mehānismu padara vēl spēcīgāku, ir tas, ka tas darbojas pāri funkciju (izsaukumu) robežām (piemēru sk. sadaļā 19.3). No otras puses, šāds funkciju izsaukumiem "paralēls" informācijas apmaiņas mehānisms ir visai resursu ietilpīgs, tāpēc nebūtu izmantojams lieki un bez vajadzības tikai kā alternatīva kontroles konstrukcijām.

```
a.5 - 3
Traceback (most recent call last):
  Python Shell, prompt 8, line 1
Syntax Error: invalid syntax: <string>, line 1, pos 3
```

Att. 19.1. Sintaktiska kļūda, nevis izņēmumsituācija

```
a = "abc"
b = int(a)
Traceback (most recent call last):
  Python Shell, prompt 10, line 1
builtins.ValueError: invalid literal for int() with base 10:
                     'abc'
```

Att. 19.2. Izņēmumsituācijas piemērs – neveiksmīga datu tipu konvertācija

```
open("nonexistant.txt")
Traceback (most recent call last):
  Python Shell, prompt 11, line 1
builtins.FileNotFoundError: [Errno 2] No such file or
                  directory: 'nonexistant.txt'
```

Att. 19.3. Izņēmumsituācijas piemērs – neeksistējoša faila vēršana

19.2. Iebūvētās izņēmumsituācijas un izņēmumsituāciju apstrādes principi

Darbs ar izņēmumsituācijām sastāv no divām daļām:

- izņēmumsituācijas konstatēšana (un paziņošana par to – izņēmumsituācijas aktivizēšana),
- izņēmumsituācijas apstrāde.

Python ir daudz gadījumu, kad izņēmumsituācija tiek konstatēta automātiski, un tās "atliek" vien apstrādāt. Šim nolūkam ir ieviesti vairāki desmiti izņēmumsituāciju kategoriju, lai būtu ērtāk noskaidrot izņēmumsituācijas cēloni, un to attiecīgi apstrādāt.

Divi piemēri ir redzami attiecīgi piemēros (Att. 19.2 un Att. 19.3):

- *ValueError* – nepareizs datu tips,
- *FileNotFoundError* – neeksistējošs fails.

Bet vēl ir daudzi citi, piemēram:

- *ZeroDivisionError* – dalīšana (vai atlikuma iegūšana, dalot) ar 0,
- *IndexError* – virknes indekss ārpus intervāla,
- *KeyError* – neeksistējoša vārdnīcas atslēga.

19.2.1. Anonīma izņēmumsituāciju apstrāde

Vienkāršākā izņēmumsituāciju apstrādes shēma ir divi secīgi bloki (Att. 19.4):

- *try* bloks – šeit tiek meklētas (lai konstatētu) izņēmumsituācijas (tātad izņēmumsituāciju konstatācija nenotiek visā kodā, bet noteiktā, iezīmētā koda daļā),
- *except* bloks (kas seko uzreiz aiz *try* bloka) – šeit tiek apstrādāta izņēmumsituācija, ja tāda tiek konstatēta,
- pēc tam neobligāti drīkst sekot *else* bloks (Att. 19.5).

```
aa = [11,22,33]
try:
    i = int(input())
    print(aa[i])
except:
    print('==unknown error==')
```

a) nav izņēmumsituācijas:

```
2
33
```

b) ir izņēmumsituācija: neeksistējošs indekss:

```
99
==unknown error==
```

Att. 19.4. Anonīma izņēmumsituāciju apstrāde bez *else* bloka

```
aa = [11,22,33]
try:
    i = int(input())
    print(aa[i])
except:
    print('==unknown error==')
else:
    print('SUCCESS')
```

a) nav izņēmumsituācijas:

```
2
33
SUCCESS
```

b) ir izņēmumsituācija: neeksistējošs indekss:

```
99
==unknown error==
```

Att. 19.5. Anonīma izņēmumsituāciju apstrāde ar *else* bloku

Izņēmumsituāciju apstrādes shēma (Att. 19.4, Att. 19.5) strādā divos iespējamos veidos:

a) izņēmumsituācija neiestājas:
   - nostrādā viss *try* bloks,
   - *except* bloks tiek izlaists,
   - nostrādā *else* bloks, ja tāds eksistē;

b) izņēmumsituācija iestājas:
   - *try* bloks nostrādā līdz kļūdas vietai,
   - nostrādā *except* bloks,
   - *else* bloks, ja tāds eksistē, tiek izlaists.

19.2.2. Konkretizēta izņēmumsituāciju apstrāde

Konkretizēta izņēmumsituāciju apstrāde papildus nodrošina:

- konstatētās izņēmumsituācijas veida noskaidrošanu,
- konstatētās izņēmumsituācijas apstrāde atbilstoši tās **veidam** un papildus informācijai, kas iegūta konstatācijas gaitā.

Šim nolūkam (Att. 19.6):

- viena *except* bloka vietā ir vairāki,

- katram *except* blokam tiek papildus norādīts datu tips, kas norāda uz izņēmumsituācijas veidu,
- datu tipam pievienojot *as object*, objektu *object* var izmantot papildus konteksta iegūšanai par izņēmumsituāciju,
- beidzamais *except* bloks drīkst būt bez datu tipa – noklusētais jeb anonīmais bloks, kas nostrādā, ja konstatētā izņēmumsituācija nav atbildusi nevienam uzskaitītajam datu tipam.

```
aa = [11,22,33]
try:
    i = int(input())
    k = int(input())
    print(aa[i]//k)
except ValueError as ve:
    print("VALUE ERROR:",ve)
except IndexError as ie:
    print("INDEX ERROR:",ie)
except:
    print('==unknown error==')
else:
    print('SUCCESS')
```

a) nav izņēmumsituācijas:

```
2
1
33
SUCCESS
```

b) ir izņēmumsituācija: nevar pārveidot par skaitli:

```
abc
VALUE ERROR: invalid literal for int() with base 10: 'abc'
```

c) ir izņēmumsituācija: neeksistējošs indekss:

```
99
1
INDEX ERROR: list index out of range
```

d) ir izņēmumsituācija: dalīšana ar 0 kā noklusētā izņēmumsituācija:

```
2
0
==unknown error==
```

Att. 19.6. Konkretizēta izņēmumsituāciju apstrāde

Kad tiek konstatēta izņēmumsituācija:

- sistēma pēc kārtas "mēģina" savietot izņēmumsituāciju ar attiecīgā bloka datu tipu,
- ja datu tips sakrīt, nostrādā tieši šis bloks,
- ja neviena bloka datu tips nesakrīt, izpildās noklusētais bloks,
- ja noklusētā bloka nav, tad izņēmumsituācija turpina būt neapstrādāta un –
  - to apstrādā kāds augstākā struktūras līmenī esošs *except* bloks,
  - vai, ja tāda nav, programma beidzas ar kļūdu.

Ja izņēmumsituācijas struktūrai tiek vēl pievienots *finally* bloks, tad tas nostrādā visos gadījumos – gan, ja tiek konstatēta kāda izņēmumsituācija, gan ja nē, un to parasti lieto ar izņēmumsituācijām saistītā bloka resursu atbrīvošanai, piemēram, faila aizvēršanai. Bloka *finally* saturu varētu vienkārši likt aiz visas struktūras, bet, liekot to speciālā blokā, tiek sintaktiski parādīta koda piederība tieši šai struktūrai (Att. 19.7).

```
aa = [11,22,33]
try:
    i = int(input())
    k = int(input())
    print(aa[i]//k)
except ValueError as ve:
    print("VALUE ERROR:",ve)
except IndexError as ie:
    print("INDEX ERROR:",ie)
except:
    print('==unknown error==')
else:
    print('SUCCESS')
finally:
    print('always FINALLY')
```

a) nav izņēmumsituācijas:

```
2
1
33
SUCCESS
always FINALLY
```

b) ir izņēmumsituācija: nevar pārveidot par skaitli:

```
abc
VALUE ERROR: invalid literal for int() with base 10: 'abc'
always FINALLY
```

c) ir izņēmumsituācija: neeksistējošs indekss:

```
99
1
INDEX ERROR: list index out of range
always FINALLY
```

d) ir izņēmumsituācija: dalīšana ar 0 kā noklusētā izņēmumsituācija:

```
2
0
==unknown error==
always FINALLY
```

Att. 19.7. Kvalificēta izņēmumsituāciju apstrāde un *finally* bloks

19.2.3. Izņēmumsituācijas manuāla konstatēšana un aktivizēšana

Darbs ar izņēmumsituācijām sastāv no divām daļām:

- izņēmumsituācijas konstatēšana (izņēmumsituācijas aktivizēšana),
- izņēmumsituācijas apstrāde.

Lielā daļā standarta gadījumu izņēmumsituācija tiek konstatēta un aktivizēta automātiski, tomēr citos gadījumos tas jādara manuāli, izmantojot komandu ***raise*** (citās programmēšanas valodās *throw*) (Att. 19.8).

```
monthnames = ['Jan','Feb','Mar','Apr','May','Jun'
              ,'Jul','Aug','Sep','Oct','Nov','Dec']

def check_month(m):
    # manuāla izņēmumsituācijas konstatācija
    if not(1<=m<=12):
        # manuāla izņēmumsituācijas aktivizēšana
        raise Exception("Month should be in range 1..12")

try:
    # šeit ievades kļūda tiek automātiski konstatēta:
    m = int(input('Input month 1..12: '))
    # šeit loģiska kļūda tiek manuāli konstatēta:
    check_month(m)
    print("Current month is:", monthnames[m-1])
except ValueError as ve:
    print('Value error:' ,ve)
except Exception as e:
    print('General exception:', e)
```

```
Input month 1..12: 12
Current month is: Dec

Input month 1..12: abc
Value error: invalid literal for int() with base 10: 'abc'

Input month 1..12: 13
General exception: Month should be in range 1..12
```

Att. 19.8. Izņēmumsituācijas manuāla aktivizēšana – gada mēneša vārdiskā nosaukuma noskaidrošana pēc kārtas numura

Tas, vai, piemēram, vērtības pārbaude uz intervālu 1..12 ir veicama, izmantojot izņēmumsituāciju apstrādes piegājienu, nav strikti noteikts, nākošajā piemērā tas parādīts, izmantojot tradicionālo piegājienu (Att. 19.9).

```
monthnames = ['Jan','Feb','Mar','Apr','May','Jun'
              ,'Jul','Aug','Sep','Oct','Nov','Dec']

def check_month(m):
    # loģiskās kļūdas apstrāde bez izņēmumsituācijas
                  izmantošanas:
    if 1<=m<=12:
        return True
    else:
        print("Month should be in range 1..12")
        return False

try:
    # šeit ievades kļūda tiek automātiski konstatēta:
    m = int(input('Input month 1..12: '))
    # šeit loģiska kļūda tiek manuāli konstatēta:
    if check_month(m):
        print("Current month is:", monthnames[m-1])
except ValueError as ve:
    print('Value error:' ,ve)
```

```
Input month 1..12: 12
Current month is: Dec

Input month 1..12: abc
Value error: invalid literal for int() with base 10: 'abc'

Input month 1..12: 13
Month should be in range 1..12
```

Att. 19.9. Izņēmumsituācijas manuāla aktivizēšana – gada mēneša vārdiskā nosaukuma noskaidrošana pēc kārtas numura (tas pats, kas Att. 19.8) – bez izņēmumsituāciju izmantošanas pārbaudei piederībai intervālam 1..12

Šajā piemērā (Att. 19.9), salīdzinot ar iepriekšējo (Att. 19.8), intervāla pārbaudes apstrāde, neizmantojot izņēmumsituāciju, izraisa to, ka kļūdas pārbaudes kods ir vairāk "sapīts" kopā ar parasto kodu (teksta "*Month should be in range 1..12*", t.i., situācijas apstrāde, notiek "iekšpusē", bez tam papildus tiek veikta pārbaude "***if*** *check_month(m)*", lai noskaidrotu, vai izdrukāt mēneša vārdisko reprezentāciju)

19.3. Lietotāja veidotas izņēmumsituāciju klases un izņēmumsituāciju apstrādes mehānisma darbība pāri funkciju robežām

Lai labāk strukturētu programmu, ņemot vērā problēmas specifiskas loģiskās kļūdas vai specifiskas situācijas, var ieviest arī lietotāja veidotas izņēmumsituāciju klases, – ir noteikts, ka tām tieši vai netieši jābūt mantotām no *Python* bāzes izņēmumsituāciju klasēm *BaseException*, *Exception*.

Nākošajā piemērā (Att. 19.11) programma rēķina nedēļas dienu datumiem intervālā no 1-1-1 līdz 4000-12-31. Datu pārbaudei tiek izmantotas šādas izņēmumsituāciju klases:

- lietotāja veidota klases *DateException* – datuma elementu (gads, mēnesis, diena) piederības pārbaudei intervālam,

- iebūvētā klases *ValueException* – ievaddatu elementu pārbaudei, vai tie ir veseli skaitļi,
- iebūvētā bāzes klase *Exception* – kontrolei, vai datumam ievada tieši trīs elementus.

Pēc koda var redzēt, ka izņēmumsituāciju apstrādes mehānisms darbojas pāri funkciju izsaukumu robežām – *try* blokā ir tikai 3 rindiņas, bet izņēmumsituāciju konstatācija un aktivizēšana notiek dziļākos funkciju izsaukšanas līmeņos (Att. 19.10).

```
try:
└──get_date()
    ├──raise Exception("Input date ... YYYY MM DD")
    ├──int(x) # automātiski aktivizē ValueError
    └──check_date_item() # atbilstība intervālam
        └──raise DateException(itemid,maxval)
except...
```

Att. 19.10. Izņēmumsituāciju konstatēšanas shēma vairāklīmeņu funkciju izsaukšanas struktūrā (kods Att. 19.11)

Lai šo pašu funkcionalitāti iegūtu bez izņēmumsituāciju apstrādes mehānisma, iespējamas divas alternatīvas:

- papildus informācijas kanālu izveidošana starp funkcijām (parametri, atgriežamās vērtības) izņēmumsituācijas informācijas komunicēšanai,
- globālu datu struktūru izmantošana izņēmumsituācijas komunicēšanai (šis gan pilnībā "neglābj" no papildus informācijas kanālu veidošanas, jo, izņēmumsituācijai iestājoties dziļākā līmenī, katrā līmenī līdz pat augšai jāievieto specifisks no funkcijas iziešanas kods).

Pēc piekļuves plašuma programmkoda struktūrai simboliski varētu ieviest **pārtraukuma operatoru** komplektu – ***break, return, raise***:

- *break* pārtrauc **kārtējo ciklu**,
- *return*, ja lietots ciklu iekšienē, potenciāli pārtrauc **visu ciklu izpildi**,
- *raise* pārtrauc **funkciju izpildi visos līmeņos**.

```
daynames = ['Sun','Mon','Tue','Wed','Thu','Fri','Sat']
monthnames = ['Jan','Feb','Mar','Apr','May','Jun',
                  'Jul','Aug','Sep','Oct','Nov','Dec']

class DateException(Exception):
    itemnames = ['Year','Month','Day']
    def __init__(self,exid,exdata=None):
        self.ExceptionId = exid
        self.ExceptionData = exdata
    def __str__(self):
        return "{} should be in range 1..{}".format(
            self.itemnames[self.ExceptionId],self.ExceptionData)

def calculate_weekday(y,m,d):
    if m<=2: # Jan,Feb regarded as 13th,14th month of prev. year
        m += 12
        d -= 1
    # typical month length is 30.6 = 28 + 2.6
    month_offset = round(m * 2.6 + 1.2) # 1.2: global offset
                   coefficient
    y1 = y // 100
    y2 = y % 100
    # next century is 5 or 6 days offset
    century_offset = y1 * 5 + y1 // 4
    year_offset = y2 + y2 // 4 # next year is 1 or 2 days offset
    return (century_offset+year_offset+month_offset+d) % 7

def calculate_max_month_len(y,m):
    if m==2:
        IsLeap = 1 if y%4==0 and (y%400==0 or y%100!=0) else 0
        return 28+IsLeap
    elif m in (4,6,9,11): return 30
    else: return 31

def check_date_item(itemid,val,maxval):
    if not(1<=val<=maxval):
        raise DateException(itemid,maxval)

def get_date(line):
    itemlist = line.split()
    if len(itemlist) != 3:
        raise Exception("Input date format should be YYYY MM DD")
    y,m,d = tuple([int(x) for x in itemlist])
    check_date_item(0,y,4000)
    check_date_item(1,m,12)
    check_date_item(2,d,calculate_max_month_len(y,m))
    return y,m,d

def main():
    line = input('Input date as YYYY MM DD:\n')
    try:
        y,m,d = get_date(line)
        wd = calculate_weekday(y,m,d)
        print("{}-{}-{}: {}".format(y,monthnames[m-
                  1],d,daynames[wd]))
    except DateException as de: # user-created exception class
        print(de)
    except ValueError as ve: # built-in exception class
        print("Date items should be integers:",ve)
    except Exception as e: # general built-in exception class
        print(e)

if __name__=='__main__':
    main()
```

```
Input date as YYYY MM DD:
1 2 3 4
Input date format should be YYYY MM DD

Input date as YYYY MM DD:
2019 12 abc
Date items should be integers: invalid literal for int() with
                 base 10: 'abc'

Input date as YYYY MM DD:
5000 1 1
Year should be in range 1..4000

Input date as YYYY MM DD:
2019 13 13
Month should be in range 1..12

Input date as YYYY MM DD:
2019 12 99
Day in this month should be in range 1..31

Input date as YYYY MM DD:
2020 2 30
Day in this month should be in range 1..29

Input date as YYYY MM DD:
2019 12 28
2019-Dec-28: Sat

Input date as YYYY MM DD:
1918 11 18
1918-Nov-18: Mon

Input date as YYYY MM DD:
2105 3 19
2105-Mar-19: Thu
```

Att. 19.11. Nedēļas dienas aprēķins ar iebūvētās izņēmumu klases izmantošanu

20. *Python* papildus bibliotēkas

Valodā *Python* bez standarta bibliotēkām ir izveidotas un piejamas daudzas papildus bibliotēkas, piemēram,
https://wiki.python.org/moin/UsefulModules
un tas ir arī viens no iemesliem *Python* popularitātei.

20.1. Trešo pušu bibliotēku uzstādīšana ar pip pakotņu menedžeri

Citu izstrādātāju veidotu *Python* programmkodu var izmantot:

- tiešā veidā ielādējot *Python* kodu no kāda repozitorija (piemēram, *GitHub*) un izmantojot lokāli savā programmā,
- instalējot vajadzīgās pakotnes (piemēram, ar rīkiem *pip*, *conda*) un pēc tam izmantojot standarta veidā (it kā tās būtu standarta bibliotēkas).

pip (rekursīvais akronīms no "pip installs packages") ir pakotņu (t.i., *Python* programmatūras vienību) pārvaldības sistēma, kas kļuvusi par de facto standartu *Python* programmatūras izplatīšanā.

Tālāk parādīts praktisks piemērs, kā instalēt pakotnes (programmatūras bibliotēkas), izmantojot *pip* (konkrētāk *numpy* un *scipy*).

Ja vajadzīgā bibliotēka nav uzinstalēta (pieejama) sistēmā, tad mēģinot to importēt, tiek parādīts kļūdas paziņojums:

```
import numpy
Traceback (most recent call last):
  Python Shell, prompt 2, line 1
builtins.ModuleNotFoundError: No module named 'numpy'
```

Pakotnes instalēšana ar *pip* notiek (operētājsistēmas) komandrindā no direktorijas (kurā pieejams *Python*) pēc šādas shēmas (*-m* nozīmē – izsaukt "kā moduli", sk. sadaļā 2.3.4):

```
python -m pip install <viens vai vairāki moduļi>
```

Piemēram, lai uzstādītu *numpy* un *scipy* bibliotēkas, jāizmanto komanda:

```
python -m pip install numpy scipy
```

Pēc konkrētās instalēšanas konsoles logā izskatīsies apmēram šādi:

```
Collecting numpy
  Downloading
                https://files.pythonhosted.org/packages/0e/c3/b
                e53614c4e3490778050e1df48fd463837297d5dd402dae3
                b500f2050eba/numpy-1.18.1-cp38-cp38-win32.whl
                (10.8MB)
     |████████████████████████████████| 10.8MB 3.3MB/s
Collecting scipy
  Downloading
                https://files.pythonhosted.org/packages/db/9e/4
                65a416eb04114e3722b17b0f4fa5235bab8a76961de51db
```

```
                    0e5850183fb1/scipy-1.4.1-cp38-cp38-win32.whl
                    (27.9MB)
     |████████████████████████████████| 27.9MB 1.1MB/s
Installing collected packages: numpy, scipy
Successfully installed numpy-1.18.1 scipy-1.4.1
```

Pēc bibliotēku instalēšanas tās var veiksmīgi importēt un izmantot *Python* vidē:

```
import numpy
numpy.__version__
'1.18.1'
import scipy
scipy.__version__
'1.4.1'
```

20.2. Bibliotēka Numpy

Numpy ir Python zinātniskās skaitļošanas bāzes bibliotēka, kuras pamatā ir **efektīvas vairākdimensiju masīvu** funkcionalitātes nodrošināšana.

Numpy bibliotēka:

- ir iegūstama caur *pip* menedžeri (sk. sadaļu 20.1),
- dažās distribūcijās (piemēram, *anaconda*) jau ir preinstalēta,
- ir pamatā daudzām citām bibliotēkām, piemēram, strādā ciešā tandēmā ar *scipy*.

Atšķirībā no *Python* saraksta list, kas funkcionāli arī var realizēt daudzdimensiju struktūras, *numpy* masīvs ir veidots tā, lai efektīvi un ērti varētu apstrādāt lielu datu apjomu:

- fiksēts dimensiju skaits un to izmēri (*shape*),
- viena datu tipa elementi,
- speciālas metodes datu apstrādei.

Tādējādi var teikt, ka pēc būtības numpy masīvs ir līdzīgs *C++* masīvam, tikai ievietots *Python* sintaksē un apgādāts ar efektīviem datu apstrādes līdzekļiem.

Līdzīgi kā objektorientētajā programmēšanā ir nerakstīts likums, ka pirmo parametru metodēs sauc *self*, tad, izmantojot *numpy*, to importē kā *np* (kas tiks izmantots arī tālākajā aprakstā):

```
import numpy as np
```

Numpy masīva datu tips ir *np.array*, un *numpy* masīvs ir izveidojams no *Python* saraksta. Numpy masīvu raksturo dimensionālā forma (*shape*) un elementu datu tips (Att. 20.1).

```
import numpy as np
aa = np.array([1,2,3])
print(aa)
print(aa.shape)
print(aa.dtype)
```

```
[1 2 3]
(3,)
int32
```

Att. 20.1. *Numpy* vienas dimensijas masīvs no veseliem skaitļiem

```
import numpy as np
aa = np.array([1,2,3],dtype=np.float64)
print(aa)
print(aa.shape)
print(aa.dtype)
```

```
[1. 2. 3.]
(3,)
float64
```

Att. 20.2. ***Numpy*** **vienas dimensijas masīvs ar norādītu datu tipu** ***float64***